FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Clock Synchronization for Many-core Processors

Filipe Miguel Teixeira Monteiro

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Pedro Alexandre Guimarães Lobo Ferreira do Souto

July 26, 2016

Abstract

Multiprocessor systems such as multi-core processors have a very relevant presence in modern lives, from computing systems such as smartphones to desktop computers.

In contrast, their adoption in real-time systems was slower but has been steadily increasing.

In recent years a new architecture for parallel processors is starting to be introduced in this strict application domain. These processors are usually called many-core processors.

Clock synchrony can be a critical feature of this type of system. Therefore, this dissertation focuses on clock synchronization for this new multiprocessor architecture, and it has the objective of implementing and evaluating a software based clock synchronization algorithm on Kalray's MPPA® - 256 MANYCORE architecture.

To determine the best approach for our problems, we provide a careful characterization of the target platform based on an experimental timing analysis of the clock source and the communication methods available on the MPPA platform.

The proposed synchronization is divided in two hierarchical levels, to accommodate for the hardware's heterogeneity. Both synchronization algorithms are adapted from the Precision Timing Protocol (IEEE 1588) used in distributed systems, but were implemented using two different programming paradigms, shared memory and message passing.

In addition to the main body of the algorithm, a delay asymmetry correction was implemented in each of the synchronization's levels in order to increase their accuracy and stability.

Furthermore, a set of experiments was done with the purpose of evaluating the quality of the implemented synchronization, and it was verified that our approach achieves good results (maximum offsets up to 5 ns inside clusters and up to 250 ns between clusters), even if it sacrifices the higher precision of hardware clock synchronization methods for the higher scalability a software implementation.

Resumo

Sistemas de multiprocessadores como os processadores *multi-core* têm uma presença muito relevante na vida moderna, em sistemas computacionais como *smartphones* ou computadores pessoais.

Em contraste, a sua adoção em sistemas de tempo-real foi mais lenta mas tem vindo a aumentar todos os anos.

Recentemente surgiu uma nova arquitetura de processadores paralelos, normalmente chamados como processadores *many-core*, que está a começar a ser introduzida neste tipo de aplicação.

Sincronismo de relógio pode ser uma característica crítica deste tipo de sistema. Assim, esta dissertação foca-se em sincronização de relógio para esta nova arquitetura de multiprocessadores, e tem como objectivo a implementação e avaliação de um algoritmo de sincronização de relógio na arquitetura MPPA® - 256 MANYCORE da Kalray.

Para determinar a melhor abordagem para os nossos problema apresentamos uma caracterização temporal da plataforma utilizada, baseada numa análise experimental aos parâmetros temporais do relógio e dos métodos de comunicação disponíveis.

A sincronização proposta está dividida em dois níveis hierárquicos, para acomodar a heterogeneidade do hardware utilizado. Ambos os algoritmos de sincronização foram adaptados do algoritmo PTP (IEEE - 1588) utilizado em sistemas distribuídos mas foram implementados usando dois paradigmas diferentes, memoria partilhada e a troca de mensagens.

Para além do corpo principal do algoritmo, foi também implementado um método de correção das assimetrias dos atrasos de comunicação em ambos níveis da sincronização para aumentar a sua precisão e estabilidade.

Adicionalmente, um conjunto de experiências foi realizado com o objectivo de avaliar a qualidade da sincronização. Foi verificado que a nossa abordagem obteve resultados positivos (*offset* máximo até 5 ns dentro dos clusters e até 250 ns entre clusters), mesmo sacrificando a maior precisão de um sincronização através de hardware dedicado em troca de uma maior escalabilidade de uma implementação em software.

Acknowledgments

Quero agradecer ao meu orientador, Pedro Souto pela motivação e ajuda que me deu ao longo de todo o desenvolvimento desta dissertação.

À equipa do CISTER por disponibilizar as suas instalações e por me permitir acesso á plataforma em que esta dissertação foi implementada. A special thanks to Borislav Nikolić for helping me "get to know" the platform in the earlier stages of this dissertation.

A todos os meus colegas e amigos dos últimos 5 anos, com o qual tive o prazer de partilhar esta caminhada.

A toda a minha família, em especial á minha Mãe, Avó e Tios, por tudo o que fizeram por mim. Sem vocês não estava onde estou nem era quem sou.

Por fim, um agradecimento muito especial para a minha melhor amiga e companheira de todas as horas, Lúcia Vaz, obrigada por toda a motivação que me deste durante todo este período de dissertação e por tudo o que me dás todos os dias. Adoro-te, 16.

Filipe Monteiro

"Time is what keeps everything from happening at once."

Ray Cummings, The Girl in the Golden Atom

Contents

List of Figures

List of Tables

Abbreviations and Symbols

AIO	Asynchronous Input/Output
AMP	Asymmetric Multiprocessor
API	Application Protocol Interface
ARINC	Aeronautical Radio Incorporated
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DAC	Delay Asymmetry Correction
DSU	Debug and System Unit
eMCOS	embedded Many-Core Operating System
FIFO	First In First Out
FLIT	FLow control unIT
FLOPS	Floating Point Operations Per-Second
FTA	Fault Tolerant Average
GALS	Globally Asynchronous Locally Synchronous
GDB	GNU Debugger
GDDR	Graphics Double Data rate
GNU	GNU is Not Unix
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
I/O	Input/Output
IA	Intel Architecture
IEEE	Institute of Electrical and Electronics Engineers
IPC	Inter-Process Communication
IPN	Inter-Processor Network
ISA	Instruction Set Architecture
KILL	Kill If Less than Linear
MIC	Many Integrated Cores
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MPI	Message Passing Interface
MPPA	Multi-Purpose Processing Array
NTP	Network Time Protocol
NUMA	Non-Uniform Memory Access
NUMA	Non-Uniform Memory Access
NoC	Network on Chip
ODC^2	On-Demand Cache Coherence
OS	Operating System
OpenMP	Open Multi-processing
PCIe	Peripheral Component Interconnect Express

PE	Processing Element
POSIX	Portable Operating System Interface
PTP	Precision Time Protocol
RFA	Reachback Firefly Algorithm
RISC	Reduced Instruction Set Computer
RM	Resource Manager
RMA	Remote Memory Access
RTD	Round Trip Delay
RTEMS	Real Time Executive for Multiprocessor Systems
RTOS	Real Time Operating System
Rx	Receive
SCC	Single-Chip Cloud Computer
SCI	Scalable Coherent Interface
SDK	Software Development Kit
SMP	Symmetric Multiprocessor
SMP	Symmetric Multiprocessor
TDMA	Time Division Multiple Access
TSC	Time Stamp Counter
Tx	Transmit
UMA	Uniform Memory Access
VLIW	Very Long Instruction Word
VM	Virtual Machine
WSN	Wireless Sensor Network

Chapter 1

Introduction

1.1 Context and motivation

This dissertation focuses on clock synchronization in real-time systems for many-core platforms.

The operation of most common systems is considered correct if the produced output values are consistent with its functional specification. This is not true for real-time systems. These systems are required not only to produce the correct results but also to do it in a timely fashion. This means that real-time systems are subject to a set of timing constraints, therefore they need to produce the expected correct results within a specified time window. Not satisfying these constraints can have consequences that range from quality losses in a non-critical service, to the complete failure of the system.

The evolution of modern technology requires even better and faster computational devices. Understandable, this extended to real-time systems.

For most common applications this need has been answered by a new type of processor architecture that we call multi-core processors. This type of processor features multiple independent processing units that can communicate with each other via a multitude of possible different mechanisms, share the same memory space, and have the objective of exploiting application parallelism to reach the desired performance increase. These processors are now the most common computer architecture. However, the adoption of multiprocessors in the real-time systems domain has been considerably slower, especially in safety-critical systems, which are subject to strict characterization processes.

These types of architectures are the next logical step for real-time systems, but there are still concerns regarding their deployment in these applications because of the unpredictability that can arise from the sharing of resources among cores.

The availability of clock synchrony among the different cores may help in increasing this predictability. Furthermore, the precision of the processor clocks can greatly impact the performance of these platforms in real-time applications

We can see the benefits of synchronized clocks in these architectures by analyzing a class of multiprocessor real-time algorithms known as semi-partitioned scheduling.

These algorithms use the concept of workload migration to optimize core utilization. In this approach, the execution pattern of a migrative task is defined at design time. This means that the execution of a task (job) can be divided in smaller pieces (sub-jobs) to be performed in multiple cores. In some algorithms, tasks will always spend the same amount of time in a given core before migrating. Thus it is important to guarantee the order of each sub-job of a task instance.

A straightforward way to enforce this order constraint is to take advantage of the available inter-process communication (IPC) mechanism. With this solution after the end of a sub-job, the scheduler in that core will send a message to the core that will execute the next piece of this instance. This sub-job will only be queued for scheduling when the message is received and processed, which means that the response time of each sub-job is heavily influenced by communication delays. In certain platforms, message contention can be rather frequent and it will result in substantial delays. It is understandable that this migration scheme can lead to a very pessimistic worst-case response time and therefore can cause low core utilization.

A better approach reduces IPC to a single message. Upon the scheduling of a migrating job, the respective scheduler can send a message to all cores that will execute the various sub-jobs. With resource to a high-precision timer and estimating the network delay of the original message, each core can appropriately schedule their respective sub-jobs. Although, the effect of communication uncertainties is reduced to the only exchanged message, the delay estimation can still lead to a substantial amount of pessimism.

A way to reduce this pessimism is to use per-core synchronized clocks. The scheduler that launches the migrative task will also send a message to all cores that will execute the respective sub-jobs, but in this scheme the message will contain a time stamp of the moment it was sent. Since every core shares the same "view" of time, the IPC delay can be measured rather then estimated. This will substantially reduce the pessimism of the scheduling and therefore improve core utilization.

Another relevant application of synchronized clocks in multiprocessor architectures is in resource management algorithms. A concrete example, PikeOS [13], a known multiprocessor real-time operating system (RTOS), implements atomic access to shared recourses trough time multiplexing. In simpler terms, each process can access a resource only at a specific predetermined time slice. To guarantee that two different processes do not access the same resource simultaneously, they need to share the same time base.

In order to address this problem, of synchronization among multiple cores of multiprocessors, a dissertation was developed last year that focused on common off-the-shelf Intel multi-core architectures[11]. We will go more in depth about this work on Section 2.3.

In recent years a new type of multiprocessor has been the target of significant research and development, the industry calls it a *many-core* processor, which received its name because it takes the focus on parallelism to the extreme by possibly harboring hundreds of processing cores, unlike their multi-core counterparts that rarely have more than a single digit number of cores. This difference brings along some key architectural changes that allow them to have such a large number of processing cores.

This dissertation will focus on software based clock synchronization for this new type of processors and it will try to make them more suitable for real-time systems.

This work was developed on Kalray's MPPA-256 many-core processor (Section 2.1.3.4), since it has some key futures that facilitate the evaluation of the proposed synchronization.

1.2 Objectives

In digital synchronous circuits, clock synchrony is usually enforced by a multitude of hardware design methodologies.

In this dissertation we take a different approach and propose to develop, implement, and experimentally evaluate a software based clock synchronization algorithm for a many-core processor architecture. The latter is crucial to assess applicability of the proposed algorithm to a given application.

Summarizing, this dissertation aims to create a way to reduce the effects of clock skew in many-core architectures, so that in the future we can take full advantage of this new hardware architecture and achieve improved performance in real-time applications.

1.3 Document Structure

The remaining of this document is comprised of five more chapters. In Chapter 2 we review some relevant state of the art concepts that constitute the theoretical and technological background of this dissertation. Chapter 3 features our definition of the clocks that are going to be synchronized and an analysis of the chosen hardware clock source. In Chapter 4 we describe the algorithms and communication mechanisms we used to synchronize these clocks. In Chapter 5 we evaluate the quality of the implemented synchronization algorithms and describe the methods that were used to extract the relevant data for their analysis. Finally, Chapter 6 presents the conclusions that can be derived from this dissertation and some future work that could be done to improve or extend the approach proposed in this dissertation.

Chapter 2

State of the art

In this chapter we present some of the relevant information about the main topics that are the focus of this dissertation.

We'll start by describing some architectural principles for multiprocessor design focusing on the most common interconnect methods and memory systems used in multiprocessor chips, we also give some examples of existing many-core processors.

The chapter is finished by a survey of some clock synchronization algorithms used in distributed systems.

A review of some multiprocessor operating systems can be found in annex A. This was done in the earlier stages of this dissertation but proved not very relevant to our work.

2.1 Many-core processor architectures

Over the years, the need for more functionalities and better performance from our computational systems was answered simply by increasing the number of transistors and clock frequency of the processors. This trend ended with the breakdown of what is referred as Dennard's MOSFET scaling law, that failed to acknowledge the effect of power leakage while in sub-threshold switching that is needed to do the necessary voltage scaling [14]. This means that unlike what Dennard and his team predicted in [15], power density is not constant and instead it increases with the miniaturization of transistors.

This problem, combined with the inability to efficiently dissipate the extra heat, led to a shift of the design approach to focus on parallelism. This change gave origin to the modern multi-core processors, where multiple independent processing cores are integrated into a single chip.

Initially these chips would only harbor a small number of cores, but over the years this number has increased to a point that some chips can have hundreds of cores. These highly parallel systems are called many-core processors. The differences between multi-core and many-core processors go beyond the number of cores. We can find very significant differences in, for example, the network used to interconnect the different cores or in the memory systems [16].

In this section will focus on some general design features used by many-core processors and give some examples of existing processors.

2.1.1 Interconnect networks

2.1.1.1 Bus Interconnection

One of the main architectural differences between multi and many-core processors lies on the way that multiple cores are connected between themselves. Historically, multi-cores communicate via a common shared bus.

Multi-cores that use this type of interconnection usually have a local memory and cache. This reduces the use of the shared bus, improving bandwidth and quality of service by reducing the amount of interference between the various cores. The bus topology presents several advantages in chips with a small number of processors because it simplifies the hardware design and helps in the implementation of synchronization features such as cache coherency protocols because of its inherent broadcast nature [1].

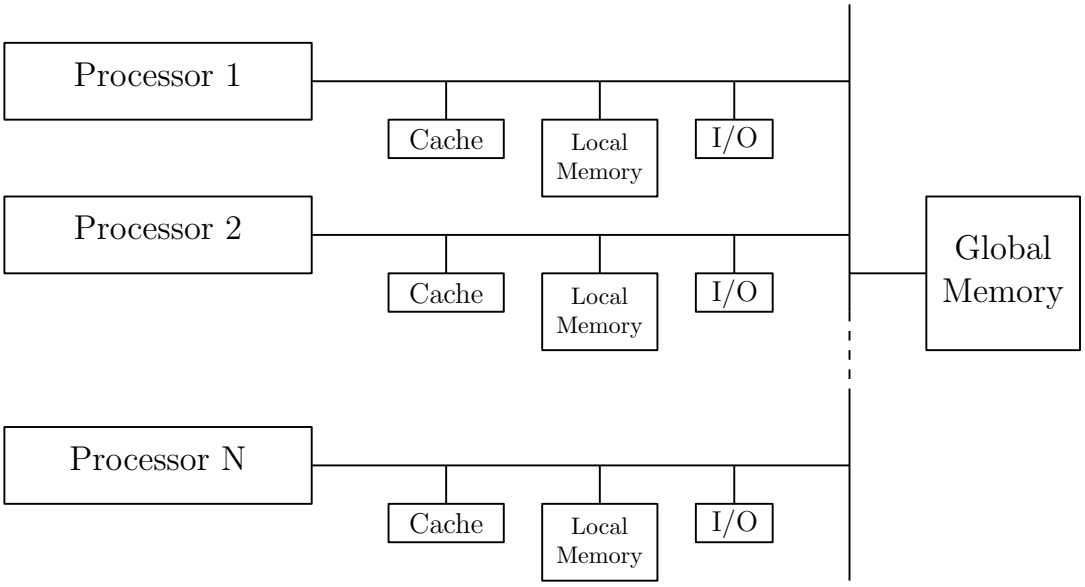


Figure 2.1: Bus connected multiprocessor [1]

2.1.1.2 Network-on-Chip

With the number of integrated cores in a single chip rising, the need for a scalable interconnect topology became a very important issue. It is clear that the bus topology was not appropriate for a large number of processors, because bus contention would lead to a significant performance degradation.

To solve this problem, the concept of *Network-on-chip* (NoC) was born. With this type of interconnection, communication tasks are done by specific NoC elements called routers [2]. These

routers can be shared by a group of multiple cores [6] or they can be a unique per-core element [5].

There several different types of NoC topologies (Figure 2.2) and the choice of one of them will greatly influence the chip's performance and scalability.

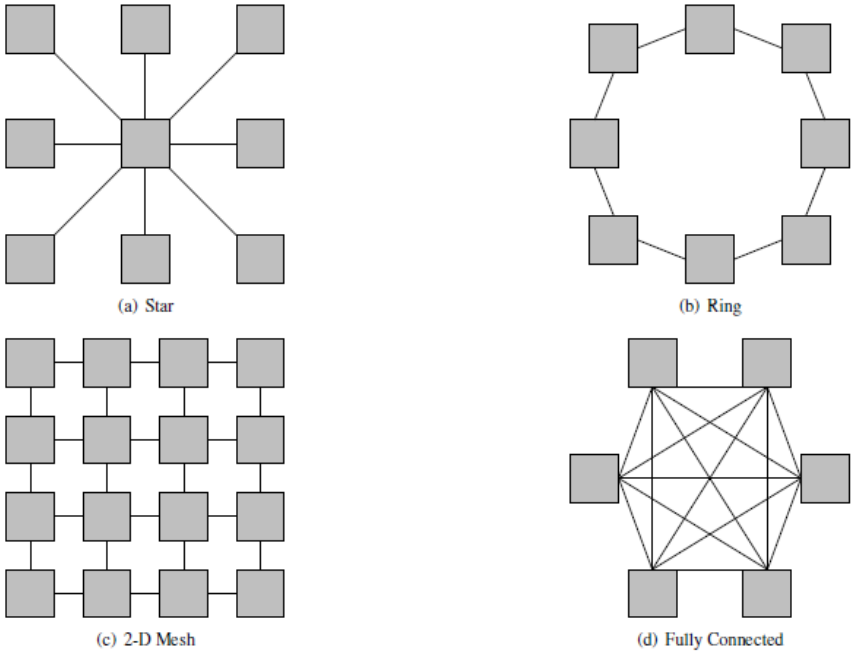


Figure 2.2: Examples of different NoC topologies[2]

Despite the large variety of different NoC topologies, the 2D-Mesh is clearly the most popular choice many-core platforms. The reason for this choice comes from the higher degree of scalability provided by this topology, since it reaches a good compromise between the total number of network links and the area used for the interconnect medium.

Therefore, let's look closer at the 2D-mesh topology. It is easy to understand that it is not possible to establish a direct connection between any two nodes, this means that there will be times that a message will need to cross multiple intermediate routers to reach its destination. This process of making a packet reach its desired destination is called routing.

Many-core development has been favoring the implementation of deterministic routing protocols. A common algorithm used in 2D-mesh NoC is the X-Y routing. This routing technique is deadlock and livelock free in two-dimensional meshes and it achieves this by limiting what kind of *turns* a packet can do in order to reach his destination (see Figure 2.3) [17]. The name of this routing algorithm comes from the fact that packets are first routed on the horizontal axis, the X axis, until it reaches the horizontal coordinate of its destination and only after this it will be routed along the vertical axis, the Y axis, until it reaches the final destination.

In the presence of traffic in specific network link a router might need to stall a packet and only send it when the necessary network link is free, this means routers will need to have some way

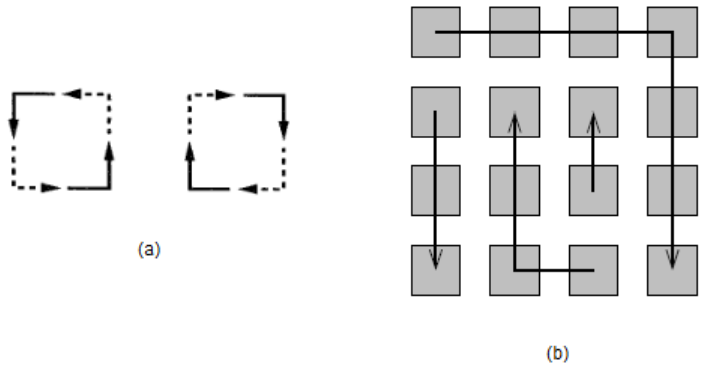


Figure 2.3: The X-Y routing algorithm; (a) The allowed turns by the X-Y routing algorithm; (b) Examples of possible packet routes [2]

to store data so no information is lost during the switching process, the process of choosing the network link where a message should sent to in order to reach its destination.

First router designs used the common store-and-forward switching method (Figure 2.4 (a)), in this approach the router ports will need to have the necessary capacity to store an entire data packet. With the increasing size of packet sizes, this data buffering became a serious challenge in router design, up to a point when became cost prohibitive in respect to the extra area needed.

In order to solve this problem different switching methods were developed, most notably what's now called as wormhole switching. In this technique a packet is divided into multiple *FLow control unITs* (FLITs) before sending, which are sent into the network in their logical order. The various FLITs then go trough the several network nodes as if they were the various stages of a synchronous pipeline (Figure 2.4 (b)). This type of switching significantly increases network throughput by increasing parallelism and can reduce the router's storage needs to the size of a single FLIT [2].

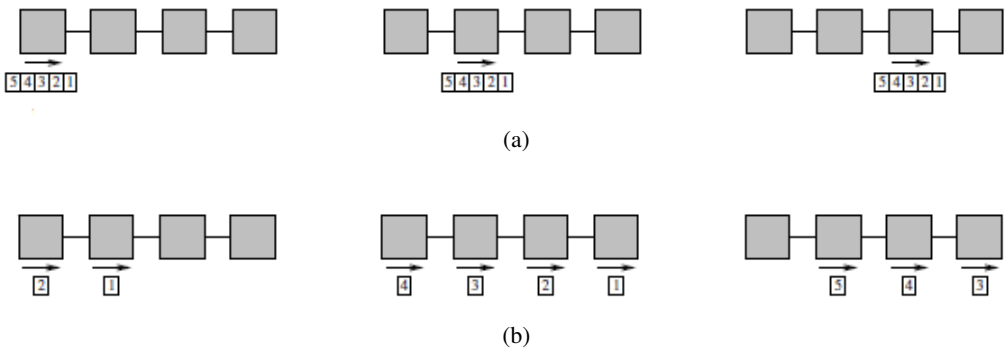


Figure 2.4: NoC switching techniques. (a) Store-and-forward switching. (b) Whormhole switching [2]

2.1.2 Memory system

Memory systems in multi-core architectures are usually implemented in one of three configurations. The most common scheme is called *Uniform Memory Access* (UMA), in this configuration any core can access any of the connected memories by using the available interconnect medium (Figure 2.5 (a)). This scheme is not common in many-core systems for not being a very scalable configuration since it creates large amounts of traffic in the *Inter-Processor Network* (IPN), which increases contention and consequently decreasing performance by increasing memory access times [18].

Another type of configuration tries to solve this scalability problem by attaching the memories directly to the processing cores, this is called *Non-Uniform Memory Access* (NUMA). In this architecture processors have direct access to their local memory bank but if they want to access any of the other memories they will have to use the IPN as it was used in the UMA configuration (Figure 2.5 (b)). It is obvious that any access to a remote memory will take considerably longer then to the local memory.

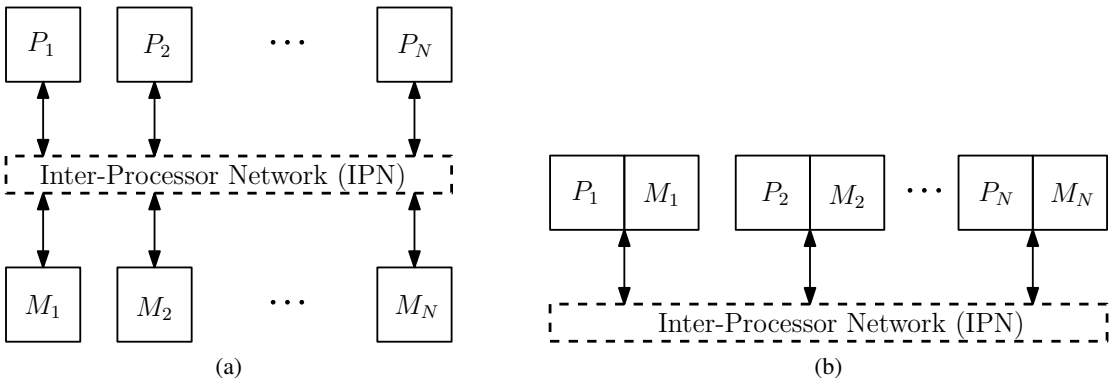


Figure 2.5: (a) The UMA multiprocessor configuration. (b) The NUMA multiprocessor configuration

While different, these two schemes share an important characteristic because they provide a global memory space where any of the processing cores can access any memory without needing the intervention of another core.

A different memory system configuration exists where this does not happen, where each processing core has exclusive access to its own private memory bank. This type of scheme is usually called a *distributed memory system* (Figure 2.6). In this configuration, remote memory access requires the cooperation of the processor that is directly connected to the memory in question. This cooperation is achieved by some sort of message passing protocol that uses the processor's IPN as the communication medium.

Many-core systems tend to favor either the NUMA or distributed configurations, or a variant of them, because of their higher scalability.

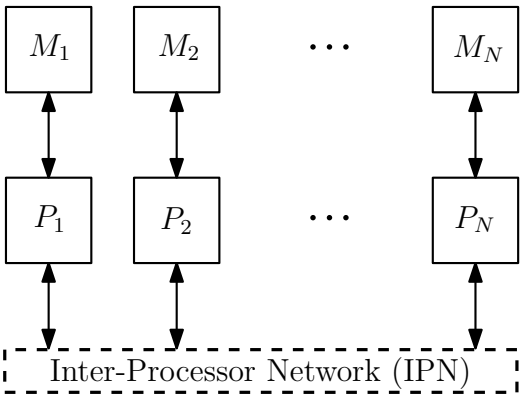


Figure 2.6: The Distributed Memory multiprocessor configuration

2.1.2.1 Cache Memory

The three configurations talked about are distinguished by the method of access to the main local memory. But in multiprocessor systems this type of memory is not the only one available. Processing cores have at their disposal a type of very fast memory called *cache memory*. The presence of this type of memory can greatly increase performance, because in large programs the same instructions can be executed repeatedly, which means certain program segments will have to be accessed several times. If instead of reading these from the main memory, the processor, can store them in cache it will lead to better performance since accessing the cache is much faster then reading the main memory.

This type of memory as a fairly limited capacity and can be divided in two different levels.

Level 1 cache, also called private cache, is a type of in-processor memory that features the fastest access times and the most limited capacity.

Level 2 cache is external to the processor and sometimes can be shared by neighboring cores, it has latency and capacity characteristics that fall between L1 cache and the main memory.

These types of memories complete the structure of a multiprocessor network node, which is slightly different in each one of the three memory configurations we already mentioned. The node configurations in Figure 2.7 are just some examples for the most common memory systems, but are in no case unique. Other structures exist since which, for example, feature shared level 2 cache memories.

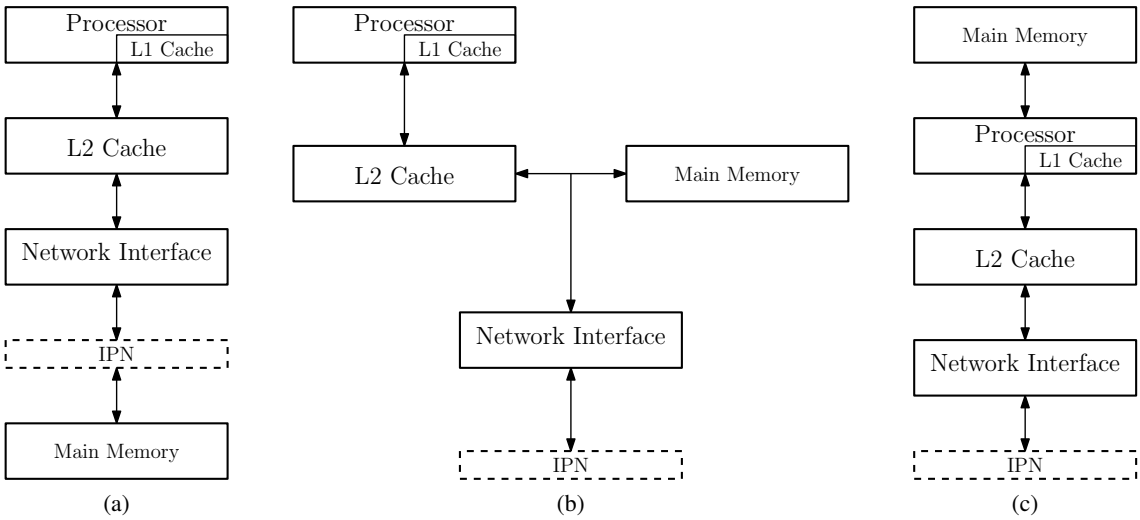


Figure 2.7: Common multiprocessor node structures. (a) UMA configuration. (b) NUMA configuration. (c) Distributed memory configuration

2.1.2.2 Cache Coherence

Lets consider a situation where two different processes share the same parameter or variable, furthermore these processes are running concurrently in different cores of a multiprocessor system. During execution, this variable will be stored in the private caches of the two processors that are running these processes. If one of them modifies this variable then all other copies will have an incorrect value. In order for this not to happen and for the system to be able to maintain a coherent state, this change needs to be propagated to any other private caches that may need it. This mechanism is called *cache coherence*.

Several types of cache coherence protocols and schemes exist, many of which require the inclusion of specific hardware modules.

The simpler and most common cache coherence protocols are the *write-trough protocol with update* and the *write-trough protocol with invalidation*. In the first, whenever a processor writes a new value in its private cache it will broadcast this value to all other cores in the system. Upon receiving one of these messages each processor will update the changed cache block with its new value if this block is present in their private cache. The second version of this protocol uses the concept of cache invalidation. Whenever a processor writes to a cache block, a broadcast is sent to notify other processors to invalidate their local copies of this block. The block is then written to the main memory so that other processes can access the updated version of the block but will only do it when they need it [18].

Many smaller multi-core systems with single-bus interconnections implement a coherence method called *snooping*. In this scheme each processor has an associated cache controller that observes all transactions made trough the communications bus. When a specific processor writes to a cache block for the first time, it makes a broadcast notifying other processors and the main

memory controller. Upon receiving this message they will invalidate their stored copies. From this moment on, the processor that made the write will be exclusive owner of this memory block and can write to it at will and without having to broadcast a notification to the other cores. If another processor wishes, it can make a request for this cache block. The main memory won't be able to respond since its value has been invalidated, but the current owner of the block will observe this request and will respond with the correct value. At this moment the processor relinquishes exclusive control over the cache block, but since the main memory is also observing the shared bus it will also update the cache block with the new value, therefore it will now be able to respond to any other read requests. This process will repeat every time one of the processors writes to a shared cache block[18].

This arrangement reduces the number coherence messages used in comparison with the write-trough protocols. Still, they don't improve much on the scalability of the cache coherence since a bus interconnect will easily become a bottleneck when we increase the number of cores in the processor.

Other coherence methodologies, such as directory based schemes and the IEEE's *Scalable Coherent Interface* (SCI) standard, present more scalable coherency methods but still can't keep up with the increase in number of cores in many-core platforms. In some cases, because of overhead created by coherence transactions, the increase in number of processing cores will lead to a drop in performance of the system[2][19].

2.1.2.3 Message Passing

Due to its poor scalability, the cache coherence methods that are used in multi-core architectures are not nearly as efficient when applied in many-core processors. An alternative to a dedicated coherence mechanism is to implement *message passing*.

In the message passing paradigm there is no shared memory space between different processes. Instead, when two processes need to share data, each one of them will have their own independent version of it and consistency can be achieved by cooperation, explicitly exchanging messages with the newest values. This means that coherence is no longer a hardware problem and is achieved trough software, shifting responsibility to the application designers.

Due to a much higher scalability, many-core development has been focusing on this technique instead of the cache coherence methods that were the norm in multi-core processors[2].

2.1.3 Examples of Many-Core Processor Architectures

In this section we'll detail some of the most known, commercially available, many-core processor architectures. More attention is given to the TILE architecture, for being a trend setter in the field, and to the MPPA® architecture since it will be the target platform of this dissertation and for its focus on real-time systems.

2.1.3.1 The TeraFLOPS Processor Architecture

The TeraFLOPS many-core processor was developed within Intel's Tera-Scale research program, and it features 80 cores, called processing engines, interconnected by an 8x10 2-D mesh network-on-chip. The entire chip was designed to operate with a clock frequency of 4 GHz, and it was manufactured with a 65 nm process.

It was designed with multimedia applications in mind, such as 3-D graphics and signal processing, it also had the objective of decreasing power consumption of the chip, particularly, by optimizing the power usage of the network routers[3].

Each network node contains two independent single-precision floating point multiply and accumulate units, 3 KB of instruction memory, 2 KB of data memory, and a crossbar router with four mesosynchronous interfaces (Figure 2.8).

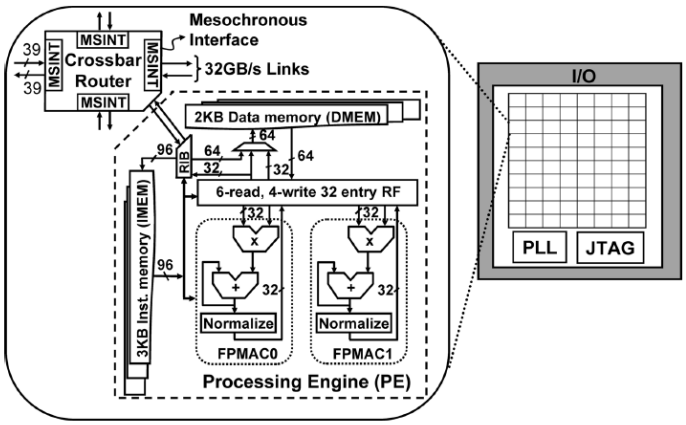


Figure 2.8: Block diagram of the TeraFLOPS processor architecture [3]

The processor's ISA defines a 96-bit VLIW that allows up to 8 operations to be issued each cycle. The instruction set is divided into 6 different types: Instructions to the floating-point units, data memory load and store instructions, NoC send and receive, the classic jump and branch instructions, synchronizations primitives(stall and wait-for-data), and sleep/wake instructions that can be used to lower power consumption dynamically. With exception of the instructions that target the floating-point units, most other instructions will execute in 1-2 cycles.

Each node features a 5-port 2-lane router that uses the wormhole switching technique. Each FLIT is divided into 6 control bits and 32 data bits, the minimum packet size is of two FLITs and there's no restriction to its maximum size. The crossbar switch has a total bandwidth of 80 GB/s.

Power consumption in the router was decreased by significantly lowering the area and number of devices used for the crossbar switch.

This architecture implements a global mesosynchronous clocking scheme that aims to increase scalability of the processor. Only one clock source is used but phase asymmetries between the clock signal in the various cores are not compensated. This means that the routers had to be designed to work with phase-insensitive communications while operations inside each core work synchronous fashion.

This processor was always an experimental endeavor with the purpose of testing the feasibility of doing floating point operations on-chip and with realistic power consumption, and to be a benchmark to validate a switched network based design for highly parallel processors[4].

2.1.3.2 The Single-Chip Cloud Computer

The SCC was Intel's second implementation of a networked many-core processor, it continues the work done by the TeraFLOPS team but with different goals. This architecture features 48 fully functional IA cores, arranged in 24 tiles, each with two Pentium P54C cores, 2 blocks of 256 KB L2 cache, 16 KB of memory to work as a message passing buffer, and the NoC router that is shared by both IA cores[4]. This processor was Intel's successful attempt to implement the data-center computing model on die.

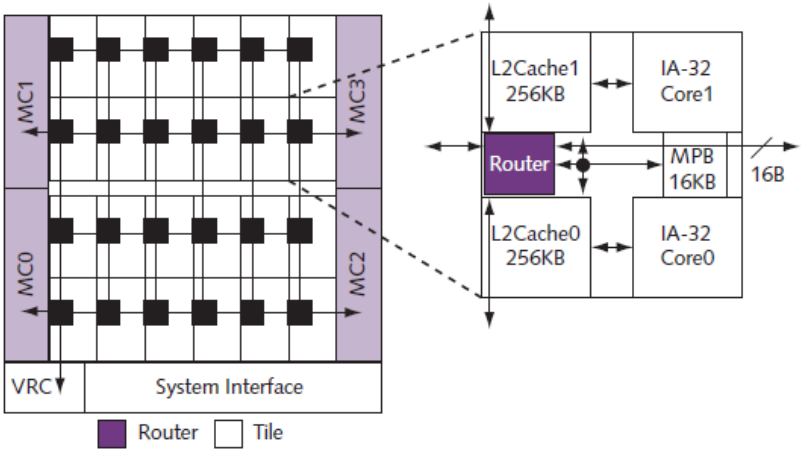


Figure 2.9: Block diagram of the Single-Chip Cloud Computer processor architecture [4]

The P54C is an augmented version of the P5 architecture that was used in the original Pentium processors, but it features a 64-bit instruction set and the necessary hooks to work in a dual SMP layout.

In the same vein as the TeraFLOPS processor, the SCC was designed to optimize power consumption and it implements both frequency and voltage scaling to do it. The mesosynchronous

clock scheme of the TeraFLOPS was omitted to increase power savings since it was considered by the SCC architects to be an overdesign. Instead a simpler scheme was used, where each tile can be running at an integer multiple of the clock source frequency that ranges from 1 GHZ to 2GHz and a clock-crossing FIFO is used to match different clock domains when needed. Clock gating is also implemented in this architecture, except within the routers.

As it happens in a lot of many-core architectures the classic hardware cache coherence mechanisms used in SMP systems were dropped for a more scalable coherence through software. The SCC implements message-passing as a way to explicitly share information between the various cores, eliminating the common shared memory programming paradigm of SMPs.

The SCC's router is heavily based of the TeraFLOPS's router, and it implements the XY routing algorithm and the already common wormhole switching scheme. Its NoC provides a bandwidth of up to 2 TB/s, much more then one P54C core could ever us. This is the reason that each router is shared by a pair of cores, aside from this, each core inside a tile behaves as if they were in different tiles.

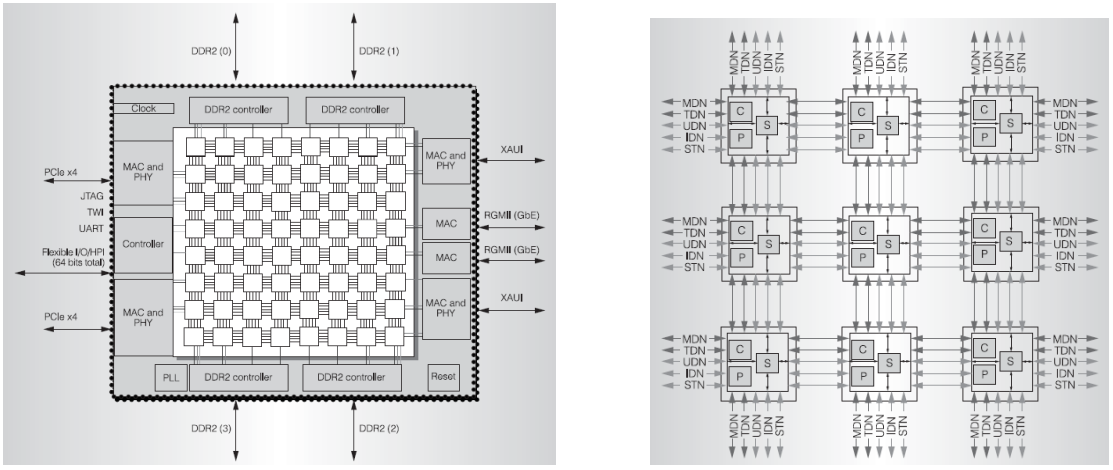
In addition to already mentioned frequency and voltage scaling, dynamic power management can be done by the system software, by turning off specific cores, tiles and even router ports.

This processor was Intel's first implementation of a fully programmable networked many-core chip that can be used for application research.

2.1.3.3 The Tile Architecture

The tile processor was developed by Tilera and was heavily based on the MIT's RAW processor that was designed by the company's founders [20].

This architecture implements a 2D-mesh interconnect topology with five independent networks, giving a total input/output bandwidth of 1.28 terabits per second (Tbps) for each core.



(a) (b)

Figure 2.10: (a) Block diagram of the TILE64 processor. (b) Array of tiles connected by the five NoCs [5].

The five networks are the User dynamic network (UDN), the I/O dynamic network (IDN), the static network (STN), the memory dynamic network (MDN) and the tile dynamic network (TDN). The two different types of classifications, static and dynamic, stem from the way data is transmitted through the networks. Dynamic networks maintain the order of messages between any two nodes, are flow controlled and guarantee reliable delivery. The static network doesn't have a packet oriented format and it allows for static configuration of the routing decisions at each tile, it can be used, for example, to send am uninterrupted stream of data between any two cores. As for the dynamic networks, each one of them serves a different purpose. The UDN is a user level network, it allows processes and threads running in different cores to communicate with low latencies providing a faster way to exchange data then through shared memory. The IDN provides direct access to the I/O devices from any of the tiles. The MDN is used to access the shared off-chip DRAM. The TDN is also used for memory management, it implements a coherent shared memory environment by allowing direct cache-to-cache data transfers [5].

As it can be seen in Figure 2.10 (b), each tile is constituted by three main hardware blocks. The internal cache unit, the processing unit and a third switch unit that handles all the network traffic coming in and out of the five networks.

So that software developers can take advantage of the interconnect medium, Tilera provides a C library called iLib that implements a set of common communication primitives on the UDN. It includes socket-like streaming channels and an MPI for ad hoc messaging between cores. It also implements different types of communication channels such as, RAW channels that have lower latencies but are reduced to the available hardware buffering and Buffered channels that have a higher overhead but allow for unlimited amounts of buffering (Table 2.1).

Table 2.1: Performance propieties of the various UDN communication methods

Mechanism	**Latency (Cycles)**	**Bandwidth (Bytes/cycle)**	**Buffering**	**Ordering**
Raw channels	9	3.93	Hardware	FIFO
Buffered channels	150	1.25	Unlimited	FIFO
Message Passing	900	1.00	Unlimited	Out of order or FIFO by key

2.1.3.4 The MPPA architecture

Kalray's MPPA-256 processors have 256 integrated user cores, 32 system cores, and feature a clustered architecture where the several processing cores are arranged in smaller groups (*clusters*) connected via two independent NoCs. In the processor's periphery there are four I/O subsystems, each one controlled by a quad-core symmetric multiprocessor (Figure 2.11) [6].

The MPPA architecture is said to implement a heterogeneous multiprocessor, because groups processors are arranged in different ways to be able to provide different types of services.

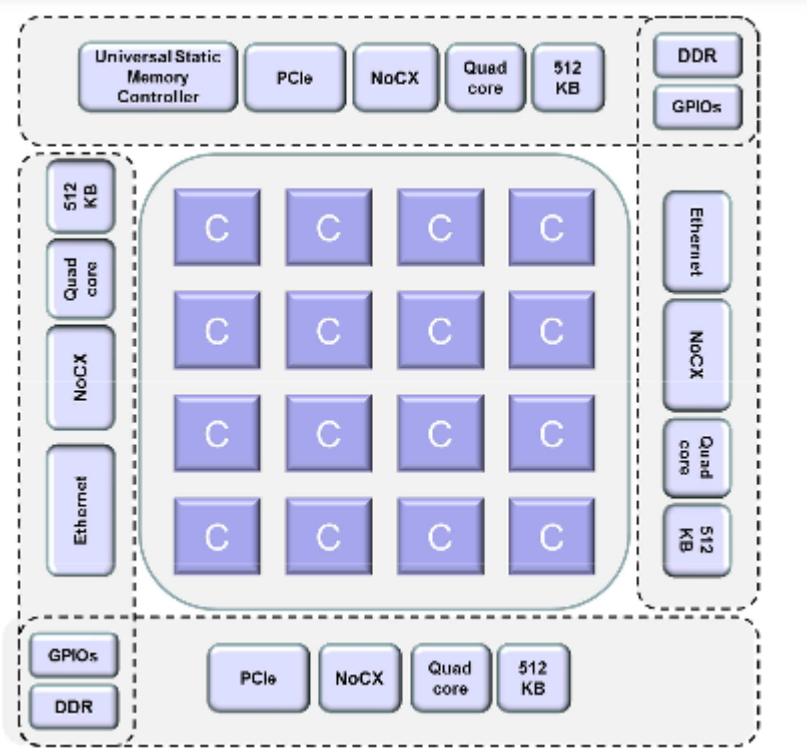


Figure 2.11: Block diagram of the MPPA-256 processor [6]

Each of the compute clusters is constituted by 16 user cores or processing elements (PE), 1 extra system core or resource manager (RM), 2 independent routers for each NoC, a Debug and System Unit (DSU) and a 2MB shared memory (Figure 2.12) [6].

The cluster's local memory is shared by 17 VLIW cores without hardware cache coherence and it's composed of 16 independent memory banks of 128Kb. Cache coherence inside the clusters is achieved by a simple software based method that the user programmer needs to pay attention to. This method will be described in more detail later on chapter 4.

This memory is divided in two sides of 8 banks that service 12 bus masters, the RM core, the DSU, both NoC routers and 8 PE Core pairs.

The address mapping of the cluster's memory can be configured as interleaved or as blocked. This has no functional implications but can have effects on performance. In the interleaved configuration sequential addresses move between memory banks each 64 bytes, making this appropriate

for highly parallel applications because of the increased memory throughput. The blocked configuration is better for time-critical applications because it reduces interference between cores since each memory bank is reserved for a single user core [21].

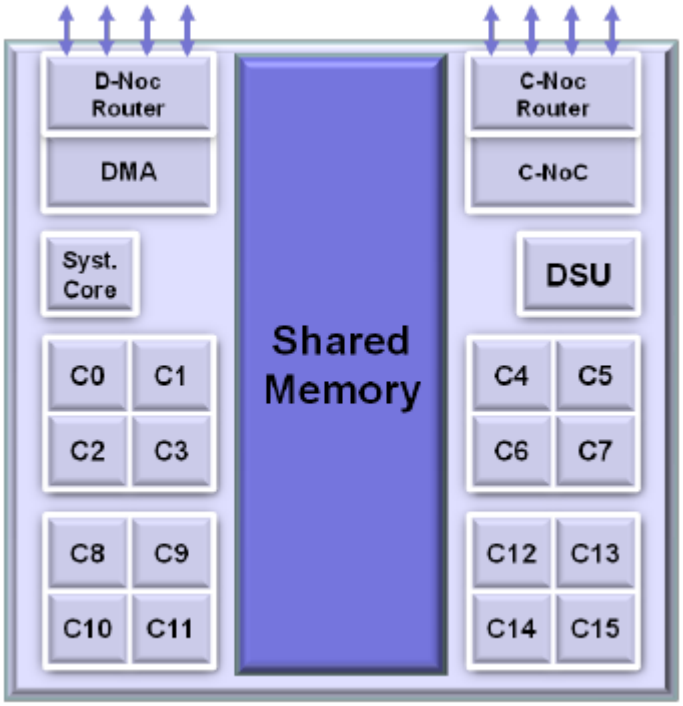


Figure 2.12: Block diagram of a MPPA cluster [6]

The MPPA cores implement a VLIW architecture with the purpose of exploiting instruction-level parallelism, since its instruction pipeline can launch up to five instructions per cycle. It has a single RISC-like ISA for both application code and system software.

A notable feature of this architecture is that it was developed with the objective of eliminating timing anomalies where a local worst-case execution does not contribute to the global worst-case, allowing a static timing analysis, of the processor and any application on it developed, to produce meaningful results [21].

As it was already mentioned, the various clusters are interconnected by the means of two independent networks, one is used for data transfers (D-NoC) and the other to exchange control information (C-NoC). Both networks share the same 2D-Torus Mesh topology and a wormhole switching style, but differ at the amount of buffering available at the cluster network interfaces [21]. Flit size is of 32-bit and the default payload data size is of 32 FLITs with a header size between 1 and 4 FLITs [22].

NoC traffic will be transparent to all but the destination node. Both networks provide guaranteed services and reliable unicast, multicast and broadcast delivery but don't implement an acknowledgment service at the packet source [23]. Hardware multicast/broadcast is only available

when targeting a group of clusters from the I/O subsystems. Multicast behavior between compute clusters is emulated by software with a series of unicast messages [22].

These networks can be accessed to from the exterior trough the NoCx extensions present in the four I/O subsystems, allowing the user to cascade multiple MPPA processors.

Since this will be the target platform of our dissertation, we will make a more in depth description of the programming paradigms for software development on Kalray's MPPA platform.

The MPPA SDK provides standard GNU C/C++ and GDB tools for compilation and debugging at cluster level. SMP Linux or RTEMS can run on the quad-core processors at the I/O subsystems and a proprietary lightweight POSIX 1003.13 profile 52 (single process, multiple threads) Operating System called NodeOS on the compute clusters. Software development can be done with two fairly different programing paradigms, a cyclostatic dataflow C based language named $\sum C$ and the more common POSIX-level programming approach[6].

In this dissertation we will focus on the POSIX-level programming. Its basic idea is that processes on the I/O subsystems will launch sub-processes on the cluster array and inside each cluster a different thread can be allocated to each one of the PE cores. The I/O subsystem spawns these sub-processes by using an adapted version of `posix_spawn` called `mppa_spawn`. In other hand the clusters can use the standard `pthread_create` combined with `pthread_attr_setaffinity_np` to start thread on specific PE cores. NodeOS also supports version 3.1 of the OpenMP standard, but maintains the 16 thread per-cluster limit [22].

The biggest difference to traditional POSIX programming and API is in the inter-process communication. IPC on the MPPA processors is done by working with special file descriptors, whose pathnames were created to identify the NoC resources used. The design of this IPC method was based on the component software model where processes are the components and file descriptors are the connectors. Multiple connectors are available to allow for various types of communication methods (Table 2.2), detailed descriptions of the software connectors can be found at [22][24] and later in Chapter 4.

Table 2.2: MPPAs IPC software connectors

Type	**Pathname**	**Tx:Rx**
Sync	`/mppa/sync/`rx_nodes:cnoc_tag	N:M
Portal	`/mppa/portal/`rx_nodes:dnoc_tag	N:M
Sampler	`/mppa/sampler/`rx_nodes:dnoc_tag	1:N
RQueue	`/mppa/rqueue/`rx_node:dnoc_tag/tx_nodes:cnoc_tag.msize	N:1
Channel	`/mppa/channel/`rx_node:dnoc_tag/tx_node:cnoc_tag	1:1

Operations with these connectors also follow the POSIX API. Common POSIX I/O functions, such as `read` and `write`, were adapted to work with the MPPA's NoCs and renamed with the `mppa` prefix. POSIX asynchronous I/O are also available for some of the connectors, and support for the `SIGEV_CALLBACK` is provided to install a callback function as notification for asynchronous operations.

The MPPA-256 processor can also be used as an hardware accelerator, for this the processor needs to be connected to the application host trough PCIe. Two special software connectors are provided to establish communication between the MPPA and the host machine (Table 2.3).

Table 2.3: MPPAs PCIe software connectors

Type	**Pathname**	**Tx:Rx**
Buffer	`/mppa/buffer/`rx_node#number/tx_node#number	1:1
MQueue	`/mppa/mqueue/`rx_node#number/tx_node#number/mcount.tsize	1:1

In the context of this dissertation is important to reference the existing support to time critical applications. Every cluster is equipped with a DSU that contains a 64-bit TSC that is addressed in the local memory and can be accessed to by any core of the cluster. The whole processor is driven by a unique hardware clock which means these counter can be considered mesosynchronous. All counters can be initialized by a specific broadcast message in the C-NoC, which results in very small offsets between the counters of the various clusters. Each core supports a lightweight implementation of POSIX timers [21].

2.2 Clock Synchronization Algorithms

This section is a review of various clock synchronization algorithms that have been developed throughout the years.

These algorithms are relevant in distributed systems where a consistent time reference is an important feature. Since each process will have its own clock derived from a local crystal oscillator it is impossible to guarantee that these have the same rate, and in systems without clock synchronization, even very small differences in frequency will eventually lead to a huge offset between two clocks.

All these algorithms share the same clock model, where each process has a counter that is incremented in an interrupt routine caused by an oscillator, this counter is the process clock. These clocks are assumed to be drift bounded by a constant value, ρ, that is known as the maximum drift rate between the clocks and real-time.

$$1-\rho \leq \frac{dC}{dt} \leq 1+\rho \tag{2.1}$$

Clocks can then be divided in three types, fast, slow or perfect clocks (Figure 2.13).

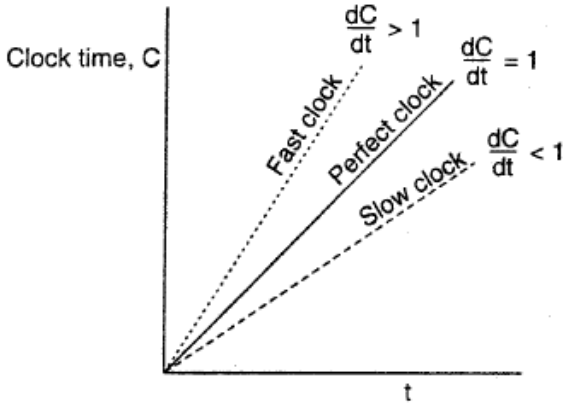


Figure 2.13: Graphical representation of clocks with diferent tick rates [7]

A given synchronization algorithm has to guarantee that, even for two clocks drifting in opposite directions, they will not ever differ by more than δ, this value is usually called the synchronization precision [7].

During this section we will present some relevant clock synchronization algorithms that have different degrees of distribution and that use assume different fault models as a consequence.

2.2.1 Probabilistic Clock Synchronization

Cristian kick started the clock synchronization approach where several clients contact a common time server and try to estimate the offset between their clocks.

Cristian's algorithm is still interesting to this day because we can reduce the error of reading a remote clock to any desired amount, which eliminates the main problem of some of the algorithms that we will discuss later on this chapter, where the read error had the significantly effect on the worst-case skews. This is accomplished by letting the slaves make several attempts to read the master's clock and calculating the maximum error at each attempt, the slave will only stop making time requests when the error reaches a desired value.

The way slaves calculate the maximum error is trough the RTD measurement technique and the read error is given by the following expression 2.2. Where ρ is the maximum drift between any non-faulty clock and U_{min} is the minimum message transit delay between master and slave [8].

$$\varepsilon = RTD(1+2\rho) - 2U_{min} \tag{2.2}$$

As it can be understood from 2.2, the error will decrease significantly with the decrease of the round trip delay. Therefore,*"each node is allowed to read the master's clock repeatedly until the round trip delay is such that the maximum read error is below a given threshold "* [8].

This algorithm is not without his faults, and it is easy to realize that this clock reading scheme will drastically increase overhead when we try to lower the skew. Another possible problem is that theres a nonzero probability of a loss of synchronization which also increases when we decrease the target skew.

2.2.2 Network Time Protocol

NTP is one of the most used protocols on the Internet. It uses a hierarchical clock synchronization approach, where nodes can be both clients or servers and are divided into different strata according to the precision of their local clocks [7][25]. It also uses different algorithm for the various synchronization strata.

At its lowest level, NTP implements the approach proposed by Cristian [26], where multiple clients contact a common time server in order to synchronize their clocks with this special node. To do this, it tries to estimate the message delay so it can compensate it and correctly adjust his clock.

The method used to do this estimation is usually referenced as the round trip delay measurement. As an example, let's consider a client and a time server that execute a round of the NTP algorithm (see Figure 2.14). The client will start the round by sending a request to the server with a time stamp T_1 of the instant this message was sent. Upon receiving this request, the time server will record its time of arrival, T_2, and send new message to the client containing T_2 and a new the time stamp, T_3, of the instant this response is sent to the client. When this second message is received, the client will have access to the three mentioned timestamps (T_1, T_2, T_3) and to a forth time stamp, T_4, of the moment this message was received. With these four time values the

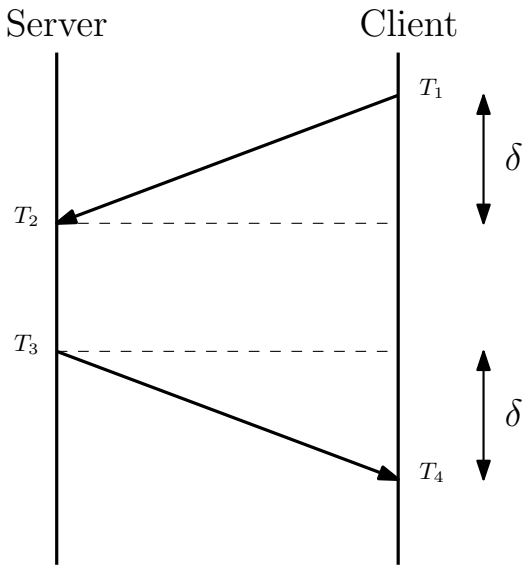


Figure 2.14: The NTP algorithm message exchange between a client and a time server

client can now estimate the communication delay (Expression 2.3), and the offset (Expression 2.4) between his local clock and the time server's, assuming the propagation delay of both messages is symmetrical, i.e., $T_2 - T_1 \approx T_4 - T_3$.

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} \tag{2.3}$$

$$\delta = \frac{RTD}{2} = \frac{(T_4 - T_1) - (T_3 - T_2)}{2} \tag{2.4}$$

2.2.3 Precision Time Protocol

The precision time protocol was standardized in IEEE-1588 and it's defined as follows:

> "This standard defines a network protocol enabling accurate and precise synchronization of the real-time clocks of devices in networked distributed systems."[27]

This standard was developed to be used in control systems, usually in a industrial environment, because it supports sub-microsecond synchronization.

Just as NTP, it also proceeds to estimate the communication delay and the clock offset during the synchronization. The main differences between the two protocols are that in PTP all message transactions are started by a master and all timestamps are taken right after sending a message or with the help of dedicated hardware. These characteristics provide greater precision to PTP but increase the number of messages transactions that need to be done in each synchronization round.

An example of a PTP synchronization round can be seen in Figure 2.15. The round starts with the master sending a multicast *sync* message to all the existing slaves. All nodes that receive this message record its time of arrival (T_2) and wait for a *Follow up* message containing the time stamp

in which the sync message was sent (T_1). Upon receiving this message the slave will send a *Delay Request* to the master and take its time stamp (T_3), the master will answer this request by sending a message with the time stamp T_4 of the instant it received the *Delay Request* message.

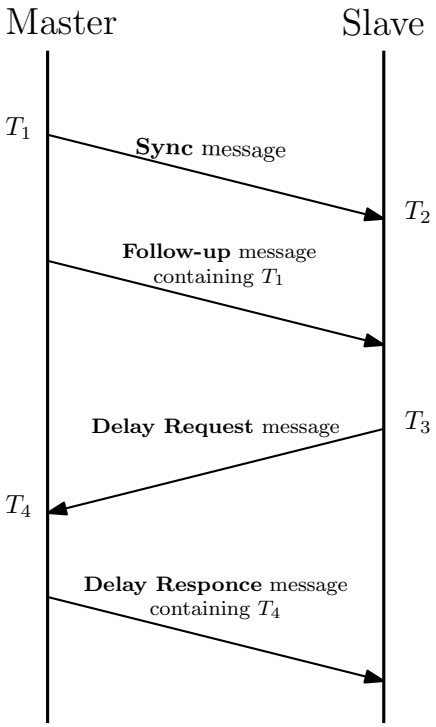


Figure 2.15: PTP message sequence chart

By the end of a synchronization round the slave clock as all four timestamps needed to estimate the clock offset (Expression 2.6) and message delay (Expression 2.5) [28].

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2} \tag{2.5}$$

$$\theta = \frac{(T_2 - T_1) - (T_4 - T_3)}{2} \tag{2.6}$$

It is worth noting that even being a centralized, PTP can be made fault tolerant by implementing an election algorithm that chooses the node best fit to become the new master clock.

2.2.4 Distributed fault tolerant algorithms

This subset of algorithms are fully decentralized since all the processes equally contribute to generate the time reference of the system, this means that there is no special process in charge of maintaining the system synchronized.

They were developed with a byzantine fault model in mind, mainly to solve the problem of *two-faced* clocks (Figure 2.16). In which a process sends different clock values to different processes. Lamport and Melliar-Smith [29] proved that it was necessary to have at least $3m+1$ clocks in order to tolerate m byzantine faults.

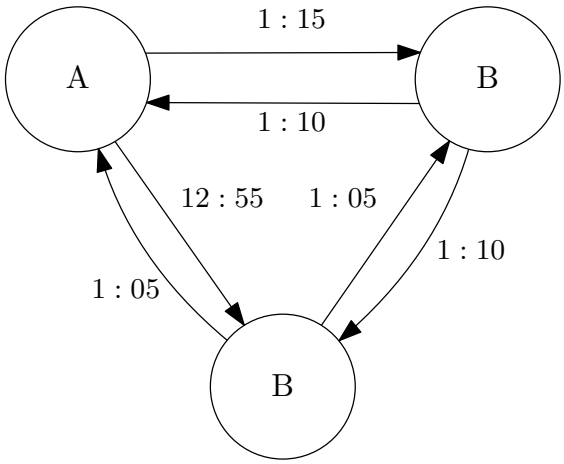


Figure 2.16: Example of a two faced clock at node A [8]

2.2.4.1 Convergence averaging algorithms

The main idea behind this type of algorithm, is to create a virtual global reference clock by calculating a *fault tolerant average* (FTA) with the clock values of all the processes that we wish to synchronize.

The creation of this reference is based on a periodic broadcast of local clock values to other processes in order for them to estimate the respective clock skews and then calculate the already mentioned averaging function [8].

In the algorithm proposed in [30], a process will broadcast its clock value periodically, it will also wait a limited amount of time, enough to guarantee it receives the broadcasts from all other non-faulty nodes in the network. After this period of time the process will calculate the already mentioned FTA in order to do the necessary corrections to its local clock. The function proposed in this algorithm, might be one of the most notable described in the literature, in which a process discards the m highest and lowest received clock values and then a common arithmetic average is applied to the remaining values. This function is able to tolerate a fixed number of m faulty clocks.

The precision of these algorithms is greatly impacted by what is called the *read error* of a clock value. This error is caused by network delay jitter and processing time fluctuations. This is

important because upon the arrival of a message, the process needs to estimate the processing and network delays associated with said message in order to make the correct clock adjustments.

To implement these algorithms there is the need for a mechanism that will guarantee initial synchronization, because they were developed with a bounded clock skew in mind. There are several solutions for this problem including the one presented by Lundelius and Lynch in [30].

2.2.4.2 Interactive consistency algorithms

The underlying objective of these algorithms is to reach a consistent clock value that all processes can agree upon. It assures agreement on the clock value of a particular sender regardless of it being faulty or not [8].

Each process will broadcast their clock value periodically, upon receiving one of these messages a process will relay it to all other processes, except to the one that sent the original message. This means that at the end of every synchronization round each process can hold up to $N-1$ clock values for the same process, N being the number of nodes in the system. After this, every process then chooses the median of these values, if the number of faulty nodes e less then a third of all processes then it is guaranteed that all non-faulty nodes will choose the same value for each sender, eliminating the problem of two-faced clocks [29].

Just like convergence algorithms its biggest limitation comes from the *read error*, but unlike them there is no need for initial synchronization for them to work correctly.

These algorithms can achieve greater precisions then their convergence counterparts for the same synchronization period but they are substantially more complex and have a higher overhead associated because they need more messages to be exchanged between the processes.

2.2.4.3 Convergence non-averaging algorithms

The best and most know example of this type of algorithm was presented by Srikanth and Toueg in [31].

In this algorithm, a resynchronization period needs to be established and every time a local clock goes trough this period the respective process broadcasts a message notifying other processes that is time to resynchronize. A process that receives this message wont immediately adjust its clock, instead, he will wait to receive at least $m+1$ messages from different processes where m is the number of byzantine faults the system can tolerate. This reassures the node that at least one non-faulty node is ready to resynchronize. When this happens, a process will adjust its clock to the next synchronization point with a small adjustment to compensate for the network delay. With enough rounds of the algorithm, all the clocks will eventually converge on the same time value.

This algorithm was presented in two different versions for the byzantine fault model. The first version assumes that authenticated messages are used for inter-process communication in order to guarantee that faulty nodes don't change received messages before relaying then and can not create fake messages claiming they received them from another process. This Authenticated algorithm

if $C = kP$ **then**
    broadcast(init, round k);
end if
if round k is accepted **then** ▷ /*Received $m+1$ (round k) messages*/
    $C = kP + \alpha$;
    relay all the $m+1$ received messages to all;
end if

Figure 2.17: Authenticated clock synchronization algorithm

can be seen in Figure 2.17, where C is the local clock, P is the synchronization period and k is the index of the current synchronization round.

The second version implements a non-authenticated algorithm and it achieves the same degree of fault tolerance simulating authenticated broadcasts resorting to the broadcast primitive of Figure 2.18. The algorithm can then be reduced to the one presented in Figure 2.19.

if Received (init, round k) from at least $m+1$ different processes **then**
    broadcast(echo, round k);
end if
if Received (echo, round k) from at least $m+1$ different processes **then**
    broadcast(echo, round k);
end if
if Received (echo, round k) from at least $2m+1$ different processes **then**
    accept(round k);
end if

Figure 2.18: Broadcast primitive for the non-authenticated algorithm

if $C = kP$ **then**
    broadcast(init, round k);
end if
if round k is accepted **then**
    $C = kP + \alpha$;
end if

Figure 2.19: Non-Authenticated clock synchronization algorithm

A clear disadvantage of this variant is that it needs at least $2m+1$ correct clocks to correctly synchronize all non-faulty processes, while the authenticated version only needs $m+1$.

Despite this, the Non-authenticated algorithm can be very interesting when considering simpler fault models, usually with crash or omission faults. For these models, the broadcast primitive can be simplified to Figure 2.20.

if received (init, round k) from at least $m+1$ different processes **then**
    accept (round k);
    broadcast(echo, round k);
end if
if received (echo, round k) from any process **then**
    accept (round k);
end if

Figure 2.20: Broadcast primitive for the non-authenticated algorithm with crash/omission fault models

This new primitive considers systems where a process can be faulty because it sometimes fails to send-receive messages or because its clock is faulty in the sense that it violates the bounded drift model mentioned earlier in this chapter. Crashes and omissions are completely transparent to the algorithm, on the other hand it can only tolerate up to m faulty clocks.

Just like the averaging algorithms they also require an initial synchronization of the system to be done [8]. The proposed approach in this algorithm is to use a system reset, which means that when a broadcast reset message is sent by one of the processes all the others will set their clocks to zero.[31]

2.2.5 Gradient clock synchronization

The gradient propriety requires that *"the skew between any two nodes' logical clock be bounded by a non-decreasing function of the uncertainty in message delay"* [32]. If we call this delay uncertainty the distance between two nodes, this means that if the implemented synchronization respects the gradient propriety than further apart nodes will have the same or larger clock skews then neighboring nodes. As we can see from Figure 2.21, each node synchronizes only with their direct neighbors, eliminating the need for a root node or a more complex algorithm that would have the need for a much higher number of messages.

The gradient propriety for clock synchronization algorithms was first introduced in [32] with the motivation for it to be implemented in wireless sensor networks, specifically in multi-hop configurations to solve several existing problems, like data fusion or the implementation of a TDMA medium access control to minimize power consumption with synchronized wake up and sleep phases. In these situations only directly connected nodes need to be tightly synchronized and therefore gradient algorithms are a good solution.

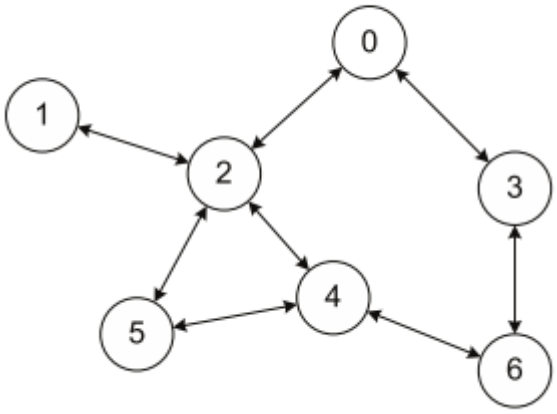


Figure 2.21: Simplistic gradient clock synchronization protocol [9]

Several algorithms have been developed with this propriety in mind, all of them with a WSN deployment in mind, where the synchronization needs are only local and nodes can only communicate directly with the neighboring nodes (Figure 2.21).

Some implement a local convergence algorithm with a common FTA [9], others use the normal communication transactions in order to achieve synchronization and save battery power [33].

2.2.6 Converge-to-Max Algorithm

This protocol is one of the simplest distributed synchronization algorithms. Each process periodically broadcasts a message with a time stamp of its local clock. Upon receiving one of these messages a process will check if the received time stamp is larger than its local clock, if this is true, then it will adjust its clock to the received time stamp. Therefore, all clocks will eventually converge to the maximum clock value on the system[34]. This algorithm guarantees a monotonically increasing clock with discontinuities and it does not have the need for any initial synchronization mechanisms.

if timeout occurred **then**
    broadcast(m_i) with local clock time stamp
    Set timeout to ϕ
end if
if message m_j received **then**
    if $m.timestamp > C$ **then** ▷ /*C is the local clock*/
        $C = m.timestamp + \alpha$
    end if
end if

Figure 2.22: Converge-to-max algorithm

The converge-to-max technique is most commonly found in wireless sensor networks and in order to increase precision, it assumes MAC-layer time stamping, to eliminate network contention times from the synchronization error.

The main shortcoming of this algorithm is in the lack of inherent fault tolerance. A fault that increases the clock value of a process will seamlessly spread throughout the entire system. The proposed method to improve fault containment, is to make each node store a list of recent timestamps for each process so it can detect outliers, erratic clock values that do not correspond to the historical evolution of that process's clock. These messages can then be discarded without disturbing the normal operation of the system.

2.2.7 Reachback Firefly Algorithm

The RFA protocol is based of a mathematical model that tries to represent the spontaneous synchronization of flashes made by firefly swarms in south-east Asia [35].

This algorithm was developed to synchronize simultaneous events in wireless sensor networks but it can be easily extended into a clock synchronization algorithm [36].

In the original model, each node will *fire* a periodic event, when another process observes this, it increases its phase shortening his own time to fire. This phase increase is calculated by a predetermined firing function. Let's look at Figure 2.23, that assumes a firing period of 100 time units, we can see that a node will immediately make a phase correction upon receiving a message that indicates the occurrence of a remote event.

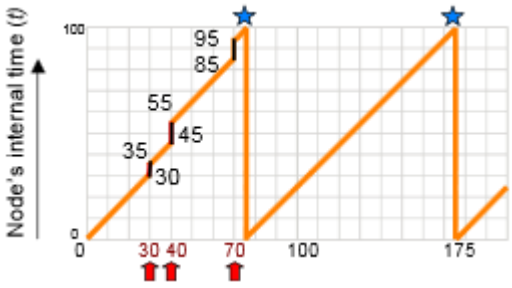


Figure 2.23: Synchronization algorithm acording with the original model [10]

RFA makes some extensions to this model in order to account for real world issues like network latency or the lack instantaneous reaction times from the different processes [10].

The first addition made is the inclusion of message timestamping at the MAC-layer to eliminate the error created by network contention, allowing to correctly identify when the sender actually fired.

The second extension is the reachback response of the algorithm. This notion was introduced because real processes cannot immediately respond to a message and these can reach out-of-order thanks to the effect of network delays. Therefore, instead of making an immediate phase adjustment when receiving a message, a node will make all adjustments after the next firing instant,

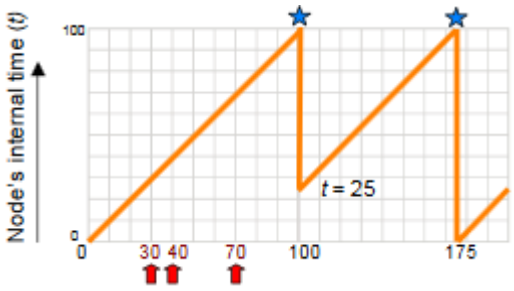


Figure 2.24: Effects of the Reachback firefly algorithm [10]

shortening the next synchronization period by the amount of all necessary corrections of the last period (Figure 2.24).

Preemptive message staggering was another addition made by RFA that tries to avoid the worst-case scenario where all nodes are closely synchronized and will all fire at the same time, creating a lot of network traffic and significantly increase contention times. By adding random delays to the moment each node will broadcast his message, this problem can be considerably minimized.

Another interesting feature of this algorithm is the simplified firing function, each jump can easily be computed by the equation 2.7 where t is the local time when the event message is received.

An important characteristic of this function is that a node will have a stronger reaction to events happening later in its own firing period. This adds an important degree of fault tolerance with respect to clocks that quickly drift away from the majority of the system and do not respect the ρ bounded drift model.

$$\Delta(t) = \varepsilon t \tag{2.7}$$

The choice of a value for ε will have a big influence in the speed ans stability of the synchronization. Larger values of ε makes the system achieve synchronism faster but might cause significant overshoot, making the system unable to reach convergence.

2.3 Clock Synchronization for Multi-Core Processors

In this section we will present a summary the work realized in a last year's dissertation by André Oliveira, a work this dissertation aims to extend [11].

André's work focused on common multi-core Intel 64 and IA-32 architectures, and it was deployed in the Linux kernel environment by the means of a loadable kernel module that allows adding code to the Linux kernel while it is running.

The first step taken during the dissertation's development, was to setup a high-definition clock in each processor core. The approach taken for this was to use the Time Stamp Counter (TSC), present in modern Intel processors, as the hardware clock source for these clocks. The TSC is a 64-bit register that stores the number of positive edges of the clock signal.

In order to make the TSC a reliable clock source for timekeeping purposes, some kernel power management options had to be disabled, such as frequency scaling, and some CPU idle states that can cause the TSC register to stop counting.

In modern processors, Intel guarantees that the TSCs of each core are synchronized, in reality the chip only has one off-core TSC that is shared by all cores, but in order to take a more general approach and because other architectures might not provide an already synchronized clock source, it was decided not to assume the existence of this synchronization, instead, this was achieved via the implemented software algorithm.

The algorithm that was developed is based on the PTP IEEE-1588 standard (Section 2.2.3). This implemented algorithm eliminates the *follow-up* and *delay responce* messages because the timestamps that are usually sent in these can be stored in shared memory and freely accessed by the slaves. This means, message exchanges carry no information but are still needed to estimate the clock offset between the master and the slaves.

Two distinct communication methods were used to implement this algorithm, which lead to different end results.

The first mechanism used were the *Inter-Processor Interrupts* (IPC). These signals are sent trough the communications bus to cause interrupts on other processors or group of processors (Figure 2.25).

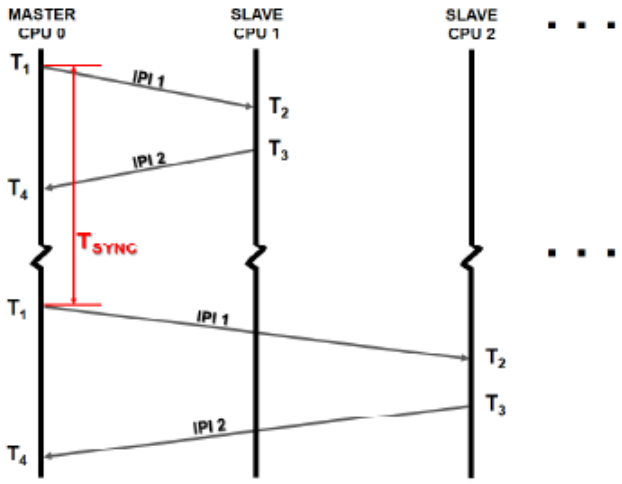


Figure 2.25: Synchronization Algorithm with IPI communication [11]

The other method to implement inter-process communication was to use the available cache coherence protocol.This was used to implement message passing by making the sender processor change a specific flag when the receiver is waiting for this change. Therefore, the receiver needs to be prepared to poll the flag when it's necessary.

In this scheme, the master core starts a synchronization round by sending an IPI to one of the slaves. This IPI is not in the synchronizations critical path, because it only notifies the slave that is time to start a new synchronization round. Upon receiving this notification, the slave will send a message to the master by setting a specific flag and will wait until it receives a response. Notice that the order of the timestamps in this adapted version of the algorithm is different from the PTP standard because the first time stamp is taken by the slave (Figure 2.26).

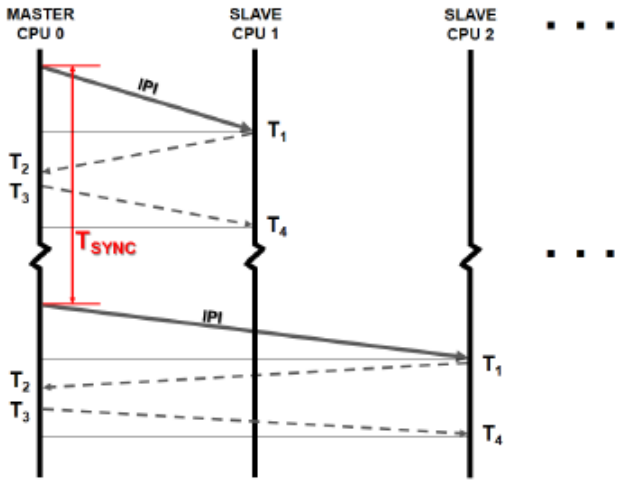


Figure 2.26: Synchronization Algorithm with Cache Coherence communication [11]

In order to assess and compare both these communication methods, the TCS's synchronization guarantee was exploited to measure the communication latency of both mechanisms. After a set of experiments it was concluded that the cache coherence method presented consistently lower latencies then its counterpart.

It was also noted that both communication methods presented a significant delay asymmetry, which can be very problematic since PTP assumes that the communication delay is symmetric to correctly estimate the clock offset. To mitigate this problem, a delay asymmetry correction method was implemented to filter out all offset samples with a delay ratio ($R = (T_2 - T_1)/(T_4 - T_3)$) that surpassed a predefined quality range. A more detailed explanation of this correction model and of its motivations can be found in Section 4.1.3.

In order to evaluate the quality of synchronization, the offset values calculated by the PTP algorithm were exported to user space along with an iteration counter for each CPU, so that the evolution of the algorithm could be analyzed. The details of this analysis can be found in [11], along with the description of all the work that was carried out last year.

In summary, we will try to extend this work by developing a software based solution, similarly to what was done here, but we will target a many-core architecture instead of a multi-core, specifically, Kalray's MPPA-256 processor.

Chapter 3

Per-Core Clock Implementation

Lets recall, from last chapter, that clock synchronization algorithms assume that each process, or core in our case, has its own local clock which is a counter usually incremented using a hardware oscillator.

In this chapter we describe how these per-core local clocks were implemented in each core of a cluster of the MPPA processor. We characterize this clock by providing a timing analysis of its read access and finish with a review of the platform's synchronization guarantees.

3.1 Clock Definition

In order to synchronize the clocks of the multiple processing cores, first we need to deploy a logical clock built in software on each of them. These clocks provide an additional layer of abstraction to the chosen hardware clock source and can be created using equation 3.1.

$$C_N = \alpha_N(HC_N - HC_N^0) - \beta_N \tag{3.1}$$

These clocks will be initialized at the start of the algorithm's execution, which means that any clock value, C_N, read from that point on is, in fact, the difference between the current value of the hardware counter (HC_N) and a constant value HC_N^0 that was the value of the hardware counter when the Nth core of a compute cluster started the algorithm.

The corrections provided by the synchronization algorithm are applied to the clocks by means of the β_N and α_N values. Where the first one is used for offset correction and the second to correct the clock's rate.

3.2 Hardware Clock Source

3.2.1 Time Stamp Counter

As mentioned in Section 2.1.3.4, each compute cluster of the MPPA processor contains a Debug and System Unit (DSU), which has a high-resolution 64-bit *Time Stamp Counter* (TSC).

In its default configuration, the TSC has the same rate as the physical clock signal, this means that the TSC will increment its value with every pulse of the underlying clock signal. If desired, frequency dividers can be applied to make the TSC run slower, but in our case we will be using the default configuration which means the clock frequency is 400MHz, giving our clocks a $2,5$ nanosecond resolution.

Each core will have to read this source in order to build its own local clock, therefore it is important to characterize its timing parameters.

3.2.2 Characterization of the Clock Source

3.2.2.1 Read access

In order to have a correct characterization of the local clock, it is important to determine how fast it can be read by one of the processing cores, and if concurrent accesses by multiple cores can cause significant interference.

Kalray provides users with a function named `__k1_read_dsu_timestamp()` to access the TSC values. Upon further investigation we found out that, this function calls an assembly instruction, called `scall 1059`, that reads a set of specific memory positions where the counter values are stored by the DSU (Figure 3.1). With this in consideration, it was questioned if directly calling the assembly instruction in the application code would make for a faster access than using the `__k1_read_dsu_timestamp()` function.

```
inline unsigned long long readTSC()
{
  unsigned long long tsc;
  __asm__ __volatile__ ("scall␣1059␣\n\t;;"
  : "+r" (tsc)
  :
  : "memory", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r11",
  "r30", "r31", "r32", "r33", "r34", "r35", "r36","r37", "r38", "r39",
  "r40", "r41", "r42", "r43", "r44", "r45", "r46", "r47", "r48", "r49",
  "r50", "r51", "r52", "r53", "r54", "r55",
  "r56", "r57", "r58", "r59", "r60", "r61", "r62", "r63",
  "lc");
  return tsc;
}
```

Figure 3.1: Code to directly read the TSC trough the assembly instruction

In order to evaluate these two operations, we developed a small program where one core successively calls the `__k1_read_dsu_timestamp()` function or the already mentioned assembly instruction, so we could do a statistical study of how long it takes to read the TSC with both approaches based on the time difference of two consecutive executions.

It was verified by executing 10000 pairs of instructions, that calling the assembly instruction takes exactly 105 nanoseconds (42 cycles) on every execution, and the `__k1_read_dsu_timestamp()`

function takes an average 142,5 nanoseconds (57 cycles), and it can range from 125 nanoseconds (50 cycles) up to 205 nanoseconds (82 cycles). These results show that a direct usage of the assembly instruction provides a faster and more consistent access to the TSC values, averaging a difference of 40 nanoseconds between the two operations.

As mentioned earlier, another important issue that should be accounted for to characterize the TSC read operation, is how concurrency affects access times. To evaluate how these values look like when multiple cores of the same cluster try to read the TSC concurrently, a program was developed where every core of a cluster successively reads the TSC.

In Tables 3.2 and 3.1 we can see the results obtained by reading the TSC simultaneously at all the 16 cores of a cluster by calling, respectively, the assembly instruction and the `__k1_read_dsu_timestamp()` function. As it can be observed, using the later function still presents larger access times, and values with an higher standard deviation.

In comparison to the earlier experiment where only one core was reading the TSC with the assembly instruction, here we can observe an average increase of 14 nanoseconds to the access times and a maximum increase of 100 nanoseconds, effectively doubling the access time. Interestingly, the average time increase when using the `__k1_read_dsu_timestamp()` function is only of 2 nanoseconds.

These results can be explained given that to read the TSC values, a core actually reads a specific set of positions in the cluster's shared memory. As it was said in section 2.1.3.4, access to this memory is done by 12 different bus masters each with its own private path to the arbiters of the memory banks. Eight of these masters are in fact PE core pairs, this means that access times will only be affected by concurrent requests made by two PE cores that share their path to the shared memory.

Table 3.1: Results obtained by concurrently calling the `__k1_read_dsu_timestamp()` function in every core of a cluster

Core ID	**Average ΔT (ns)**	**Maximum ΔT (ns)**	**Minimum ΔT (ns)**	**Standard Deviation σ**
0	143,8	207,5	125	7,63
1	146,1	232,5	125	8,39
2	143,7	202,5	125	7,42
3	143,8	210	125	7,55
4	144,1	205	125	7,79
5	143,7	205	125	7,66
6	143,6	202,5	125	7,65
7	143,7	212,5	125	7,57
9	143,6	195	125	7,41
10	143,8	262,5	125	7,89
11	143,8	185	125	7,50
12	143,6	202,5	125	7,51
13	143,8	190	125	7,60
14	143,5	210	125	7,51
15	143,4	200	125	7,44

Table 3.2: Results obtained by concurrently calling the assembly instruction in each core of a cluster

Core ID	Average ΔT (ns)	Maximum ΔT (ns)	Minimum ΔT (ns)	Standard Deviation σ
0	118,5	157,5	105	5,70
1	121,7	172,5	105	6,66
2	118,4	165	105	5,51
3	118,8	155	105	5,54
4	118,5	185	105	5,71
5	118,7	187,5	105	5,30
6	118,2	160	105	5,53
7	118,5	175	105	5,37
8	118,5	177,5	105	5,75
9	118,7	162,5	105	5,55
10	118,7	210	105	5,70
11	118,0	150	105	5,27
12	118,3	187,5	105	5,73
13	118,5	155	105	5,29
14	118,4	165	105	5,26
15	118,7	177,5	105	5,71

3.2.2.2 Timer Precision

Another interesting feature that should be characterized is the precision of the platform's timed events, i.e, timers. While this might not be directly related with the clock source it is relevant since for many clock synchronization algorithms the precision in which a process can lunch an event can greatly impact the synchronization's quality. For example, in an algorithm like PTP, the timer's precision won't have much effect on the synchronization's precision, but in an algorithm such as the one in Section 2.2.4.3, this characteristic is actually relevant and has a large impact in its precision. Therefore, this analysis is important so we can make a better choice of which algorithm should be implemented.

Within the cluster array of the MPPA processor, the only way to lunch timed events is by using the classic POSIX timers. A callback function can be installed in these timers, using `SIGEV_CALLBACK`, so that when they expire this callback function is triggered.

In order to evaluate the precision of MPPA's POSIX timers a small application was written where we setup a periodic timer with an associated callback function. In this callback we read the TSC as soon as possible, so that we could verify the offset between the desired timer period and the actual time between two triggers of the callback function.

After 10000 iterations with a period of 10 ms, the jitter ranged from 0 to 342,5 ns and had an average value of 105,35 ns. This means that an algorithm which relies on precisely timed events will have these values as an upper bound on its precision.

3.2.2.3 Native TSC synchronization

The Kalray MPPA-256 architecture was developed with the objective of reducing timing anomalies to a bare minimum, with this in mind the entire processor is driven by a single hardware clock. Because of this design choice, it is guaranteed by Kalray[21] that the TSC's of all clusters are mesosynchronous, meaning that all counters share the same rate but might have unknown offsets.

To reduce these offsets all counters need to be initialized by a hardware supported broadcast message on the C-NoC made by the I/O subsystem to the compute clusters, which will result in offsets *"of about a dozen cycles"*[21].

It's important to note that this broadcast needs to be made by an I/O subsystem because hardware broadcast is not supported between compute clusters but only when one of the I/O subsystems targets multiple clusters.

To test these claims, we developed a small application that tries to read the TSC in all clusters at the same time. Firstly we use a C-NoC hardware broadcast, supported by the `Sync` connector (Section 2.1.3.4), together with the `__k1_write_dsu_timestamp()` function to initialize all TSCs just as described before. After this, we try to sample all counters simultaneously, again, taking advantage of the already mentioned hardware broadcast with the `sync` connector to implement a master-slave barrier synchronization between all clusters and the I/O subsystem. The operation of this barrier can seen in Figure 3.2, where the I/O subsystem (master) waits for all clusters (slaves) to notify him that they are ready to sample their TSC. After sending this notification they will block until receiving a message from the master, but the master will only respond after receiving a notification from 16 slaves. This response will be made trough the already mentioned hardware supported broadcast so that all slaves unlock and read their TSCs at the "same" time.

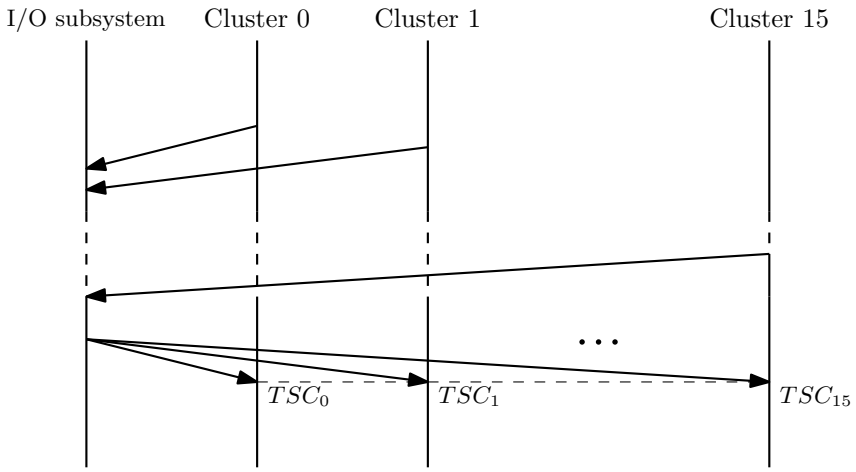


Figure 3.2: Master-slave barrier operation to periodicly sample of the TSC

After 5 hours of execution, the maximum offset of any two clusters remained small and did not show any kind of growing trend. The maximum observed offset was of 188 cycles, this value can be attributed to delay differences that might occur during the barrier operation and to the timing

uncertainties of the read access. Therefore, we can assume that the claim in [21] is valid and that we can take advantage of it to measure any relevant timing characteristics of the platform, which will prove very helpful in evaluating the quality of synchronization that we propose to implement.

As mentioned earlier, our logical clocks are based on the TSC, but despite of this, we decided not to assume that they are synchronized for the purpose of this dissertation, instead synchronicity is to be achieved by the execution of a software based algorithm that we will describe in the next chapter. This decision stems from the fact that other many-core processors, whose design was not focused on timing predictability, do not provide access to a synchronized clock source, some even adopt a GALS (Globally Asynchronous Locally Synchronous) type of clock, where different asynchronous time domain coexist in the same chip, making the issue of clock synchronization to be even more relevant.

Chapter 4

Clock Synchronization

In this chapter we will present how we achieve system wide synchronization with an hierarchical scheme that takes advantage of the hardware's hierarchy. Therefore, the synchronization is implemented at two different levels: between the 16 cores inside each cluster and then, at a higher level, between the 16 clusters of the MPPA processor.

4.1 Intra-Cluster Synchronization

In this section we take a closer look at the clock synchronization that was implemented between the 16 PE cores of each compute cluster, starting with a description and analysis of the available communication mechanisms inside the clusters, followed by a description of the implemented algorithm and the filtering method used to correct the communication delay asymmetry.

Our synchronization approach is an adapted version [11] of the common PTP algorithm (Section 2.2.3) used in distributed systems.

4.1.1 The Communication Methods

4.1.1.1 Cache Coherence and Data Transfers

In addition to the scalability issue presented in 2.1.2.2, multiprocessor architectures designed for real-time applications, like the Kalray MPPA, usually don't implement hardware based cache coherence methods since they can be fairly unpredictable, making it harder to carry out a meaningful timing analysis.

Whereas between clusters, coherence is achieved trough the message passing paradigm, inside each cluster, coherence of shared data is enforced by Kalray's NodeOS via a process of cache invalidation and write buffer purging that is dependent on the application programmer [22].

This technique achieves a compromise between performance and the desired predictability for real-time applications.

In this cache coherence model, access to shared data is always a critical section and needs to be protected by a lock (Figure 4.1). NodeOS supports the common POSIX API for parallel

```
—Access to private Data only—
lock(critical_section);        ▷ Data cache invalidation

    —Access to shared Data—

unlock(critical_section);      ▷ Data cache flush
```

Figure 4.1: Pseudocode of the protected access to shared data

programming, therefore when, for example, a core locks a mutex it enters a different mode of operation called shared mode, it's important to notice that this also happens when we use other POSIX synchronization primitives such as barriers, semaphores, etc.

This change in operation mode is done by invalidating the local data cache, which means all data read during this mode will be fresh of the cluster's shared memory bank. For this reason, access to shared variables protected by this lock will now be safe, since the core has their latest value and no other core can access it until the primitive is unlocked. When this happens the core's private cache will be flushed into the shared memory and the core will go back to the normal mode of operation. This mechanism will guarantee that every piece of shared data always has its freshest value on the memory banks and that any core trying to read these shared variables won't be accessing outdated information, as long as the synchronization primitives are used correctly.

This method will be transparent to the implementation of our synchronization algorithm and is used only for exchanging the necessary timestamps of the PTP algorithm between PE cores.

4.1.1.2 Message Based Communication

Several clock synchronization algorithms, PTP included, are based on the idea of exchanging synchronization messages and time stamping the instants of sending and receiving them to estimate clock offset.

To implement message based communication between cores inside a compute cluster, we have two different methods available, POSIX signals and NodeOS events. These methods are different in their nature but they share an important characteristic, both are only a notification/synchronization mechanism, which means neither of them can be used to carry data between cores.

NodeOS implements only a subset of all POSIX signals, including the `SIGURS1` and `SIGUSR2`, which are user defined signals. Using the `sigaction` struct we can install a custom signal handler function for these signals, that will trigger when a core receives one of them. This communication method is therefore completely asynchronous and non-blocking [22].

The second method is specific to the Kalray MPPA architecture and to their proprietary operating system NodeOS. The so called NodeOS events use special hardware features to establish communication and synchronization between cores. The API for these events is constituted by two functions, `nodeos_event_send` and `nodeos_event_receive`, and provides 32 different possible events that can be used in any core. Since each PE core can only execute one user thread, the send function uses the `pthread_t` identifier to address the destination of a specific

event. As for the receive function, it can be configured to work in two ways, either blocking or non-blocking. If configured as blocking the function will only return upon the reception of a new event, otherwise, if it is configured as non-blocking it will return `NODEOS_EVENT_UNSATISFIED` if the event has not yet arrived at the moment it was called and must be polled in order to detect the arrival of a new event [22].

It is important to note that events are not queued, which means if an event is sent multiple times before a call to the receive function is made, it will have the same effect as the arrival of a single event.

To compare these two communication methods we wrote a small program where the PE0 core (Master) exchanged 10000 message pairs with each one of the other cores inside the cluster (Slaves).

In Tables 4.1 and 4.2 we present a summary of the results of this experiment for POSIX signals and NodeOS events respectively. They show that POSIX signals have considerably larger delays than NodeOS events, which should not be surprising since the events are tied to special hardware features that were specifically designed to efficiently synchronize the multiple cores inside a cluster, which contrasts with the POSIX signals that are part of a high level API with no concerns for real-time behavior.

Table 4.1: POSIX signals latency experiment results

Core ID	**Average Delay M⟶S (*ns*)**	**Average Delay S⟶M (*ns*)**	**σ M⟶S**	**σ S⟶M**	**Average Delay Asymmetry (*ns*)**	**σ Delay Asymmetry**
1	3160,9	4158,0	37,25	21,17	997,1	45,12
2	3122,4	3811,2	34,48	31,62	688,79	48,07
3	3130,2	3794,8	33,29	29,10	664,68	31,52
4	3108,2	3798,3	35,27	31,58	690,19	43,45
5	3113,8	3797,4	32,05	27,51	683,68	30,52
6	3141,5	3804,3	34,63	34,83	662,88	36,60
7	3115,1	3802,4	31,19	25,45	687,46	32,11
8	3112,7	3806,6	35,40	30,58	694,07	39,40
9	3114,0	3820,4	31,93	30,26	706,41	25,72
10	3133,2	3845,5	34,25	32,51	712,38	32,44
11	3130,9	3817,1	32,16	31,60	686,19	29,31
12	3110,6	3806,7	35,69	30,95	696,13	38,80
13	3121,7	3816,1	32,67	33,51	694,47	23,54
14	3143,1	3826,9	32,52	34,08	683,82	38,79
15	3101,1	3802,2	29,85	28,70	701,20	23,07

In this dissertation we define delay asymmetry as being the absolute difference between the "Master to Slave"(M⟶S) and "Slave to Master"(S⟶M) delays. This value is very important when we consider the implementation of a PTP based algorithm, since it assumes symmetric delays in order to successfully estimate the offset between clocks.

Another important note that can be taken from the experiments is that NodeOS events present much higher predictability, with much lower delay standard deviation and lower delay asymmetry than POSIX signals.

Table 4.2: NodeOS events latency experiment results

Core ID	Average Delay M⟶S (*ns*)	Average Delay S⟶M (*ns*)	σ M⟶S	σ S⟶M	Average Delay Asymmetry (*ns*)	σ Delay Asymmetry
1	902,15	810,50	4,11	1,77	91,65	4,32
2	869,28	804,55	2,84	3,26	64,7	4,47
3	928,68	800,71	2,34	1,95	197,97	3,00
4	888,62	804,94	1,19	1,43	83,68	1,75
5	856,75	799,18	2,90	1,46	57,58	2,52
6	862,34	802,67	1,54	1,99	59,67	2,34
7	847,15	801,02	0,77	1,74	46,13	1,53
8	864,06	809,25	1,33	0,80	54,81	1,82
9	870,15	805,62	0,93	1,55	64,53	1,91
10	865,72	814,09	2,20	0,48	51,63	2,25
11	857,10	809,91	2,00	1,66	47,20	3,63
12	879,81	807,5	0,26	0	72,31	0,26
13	930	805	0	0	125	0
14	872,40	815	1,58	0	57,40	1,58
15	862,5	810	0	0	52,5	0

From Figure 4.2 and the raw data presented before we can observe that NodeOS events provide a faster and more reliable method for our application, since the communication stands in the critical path of the precision of any clock synchronization algorithm that is to be implemented. Next we will discuss how these methods can be used in the implementation of a PTP based synchronization algorithm.

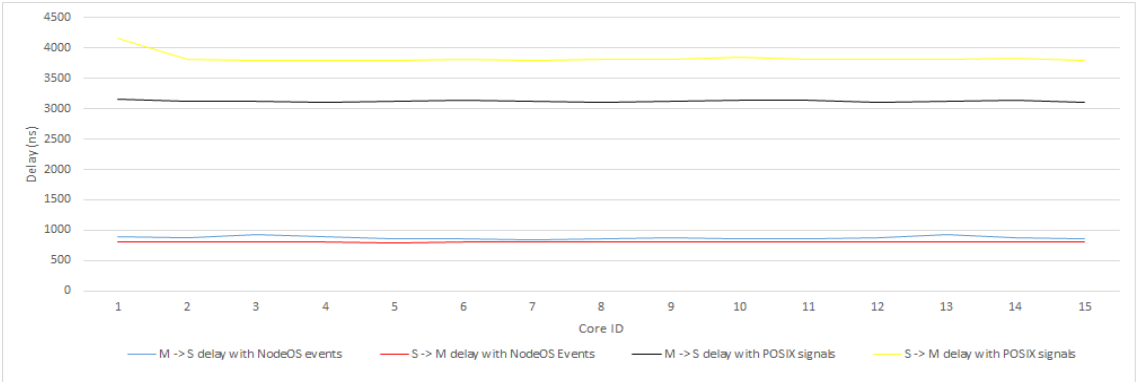


Figure 4.2: Average Delay communication inside the compute clusters

4.1.2 The Synchronization Algorithm

As it was mentioned at the start of this section, we decided to synchronize the clocks in each core of the same cluster through an algorithm adapted from the Precision Timing Protocol (PTP), defined in the IEEE 1588 standard (Section 2.2.3).

The adaptations to the standard PTP were proposed in [11] to be used with the cache coherence communication method described in section 2.3. We chose this approach because clusters have

a shared memory architecture very similar to the multi-core systems that were the algorithm's original target platform. Another reason stems from the blocking nature of the communication method, a characteristic that is also shared with the original work.

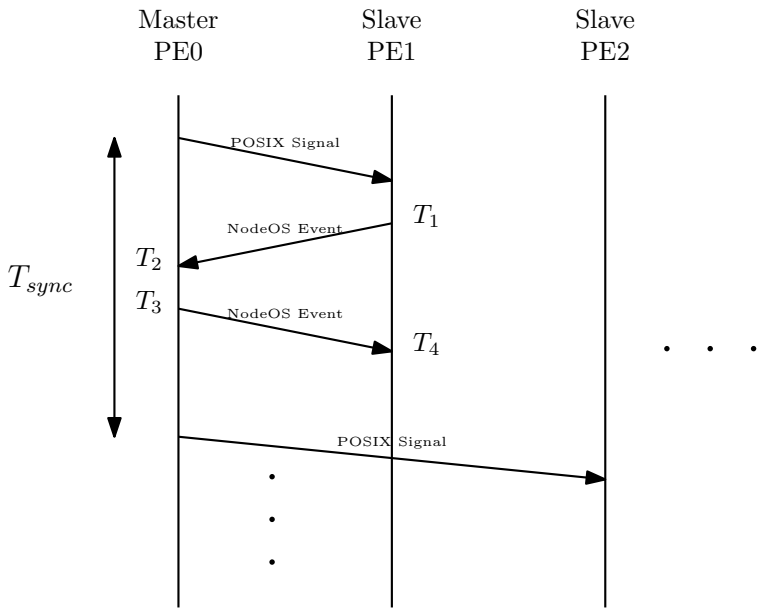


Figure 4.3: Message sequence chart of the intra-cluster Synchronization

In this implementation, the *follow up* and *delay responce* messages were eliminated, since the synchronization has a shared memory space that can be used to store and access the timestamps that otherwise would need to be sent in these messages.

Because NodeOS events do not provide a way to do asynchronous communication we use a POSIX signal to notify the slaves to start each synchronization round, this signal is not in the synchronization's critical path since it is not used to estimate the clock offset between the two cores. For this task, we use the NodeOS events as a communication method.

After one of the PE cores receives the notification signal it will immediately send an event to the master, in our specific case we use `NODEOS_EVENT_0` for this message passing, meaning that the other 31 events are still free to be used in a user application. Both instants of sending and receiving this event are timestamped (T_1, T_2) and stored. After receiving this message, the master will send another event to the slave, and again both sending and receiving of this message are timestamped (T_3, T_4) (See Figure 4.3). After this, the four timestamps are used to estimate the clock offset between master and slave.

For the slave to be able to access the master side timestamps (T_2, T_3) correctly, the master will lock a mutex that protects the access to the shared variables where these timestamps will be stored before sending its message and will only unlock it after properly storing the correspondents timestamps in each variable. In the other hand, the slave will try to lock this same mutex immediately after receiving the message from the master, but it will only be able to do it once the master releases it. With this, we guarantee that the slave will only have access these timestamps after

the master flushes its local cache, that contains the T_2 and T_3 timestamps, into the shared memory bank.

$$\theta = \frac{(T_4 - T_3) - (T_2 - T_1)}{2} \tag{4.1}$$

As mentioned in section 2.3, because the direction of the messages is inverted with respect to the PTP standard, therefore the offset calculation changes to expression 4.1.

4.1.3 Delay Asymmetry Correction

The method of estimating the clock offset used by the PTP algorithm relies on the assumption that communication delays are symmetric, i.e., delays from "master to slave" ($\delta_{m\to s}$) and from "slave to master" ($\delta_{s\to m}$) are the same. As it can be seen from expression 4.2, if these delays are not symmetric, then the offset calculated by the algorithm (θ_{PTP}) will not be the real offset (θ_{Real}) between the two clocks and can have a rather large error associated.

$$\begin{aligned} \theta_{PTP} &= \frac{(T_2 - T_1) - (T_4 - T_3)}{2} = \frac{(\theta_{Real} + \delta_{m\to s}) - (-\theta_{Real} + \delta_{s\to m})}{2} \\ &= \theta_{Real} + \frac{\delta_{m\to s} - \delta_{s\to m}}{2} \quad \wedge \quad \delta_{m\to s} \neq \delta_{s\to m} \Rightarrow \theta_{PTP} \neq \theta_{Real} \end{aligned} \tag{4.2}$$

In order to minimize this effect and achieve higher synchronization precision, we implemented a Delay Asymmetry Correction (DAC) model based on [37] and with the adaptations made in [11].

This correction method does not change PTP's basic message exchange and it only influences the way data is processed after each synchronization round. The main idea of this model is to filter out possible "bad samples". To do this we estimate the delay asymmetry between master and slave, trough a ratio of the PTP timestamps ($R = (T_2 - T_1)/(T_4 - T_3)$). This estimation will be of greater precision when the clock offset between master and slave is smaller than the communication delays.

One of the reasons that lead us to use this model is because it does not add extra layers of complexity to the main body of the algorithm, unlike other asymmetry correction methods [38].

As it can be seen from Figure 4.4, only samples that pass the filter condition will be used to correct the slave's clock. In the original model[37], these sample would not be automatically applied and instead would be fed into a 2nd stage filter. In our case we omit this stage because the Kalray MPPA architecture has a single hardware clock source, which guarantees that the clock rate is the same in every core. Therefore, the 2nd stage filter would only be a source of error by creating drift between the clocks instead of minimizing it. Any other samples that don't respect the initial filter condition will be immediately discarded.

Because of this lack of clock drift, another change was made to the original filter is the inclusion of an adaptive filter condition. We do this by starting with a larger acceptance interval ($[80\%, 120\%]$) that will be decreased as samples are accepted so that only accept better samples than the ones that were already applied, eventually only samples with ratio of 100% will be accepted.

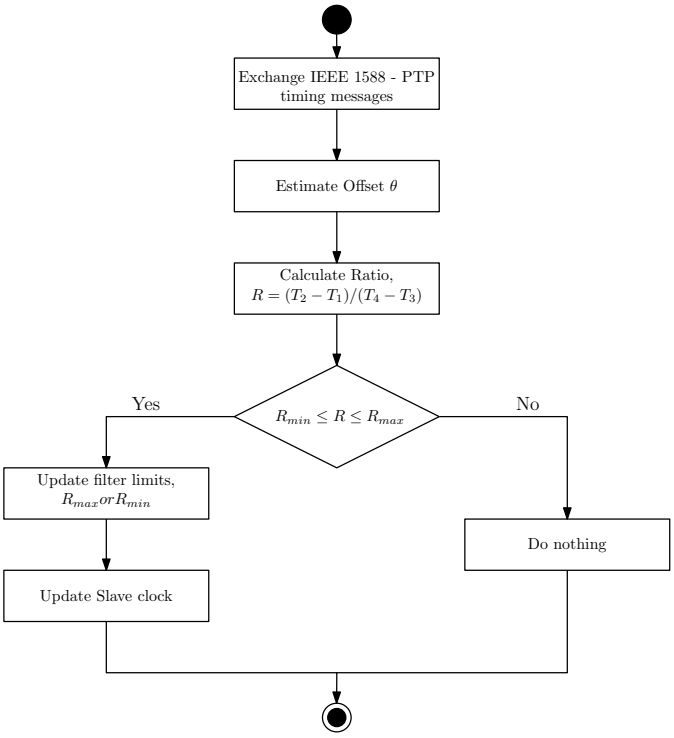


Figure 4.4: Flowchart of the adapted DAC model for the intra-cluster synchronization

These adaptations were made in [11], because its target platform was also drift free, the same reason why we adopted these changes in this dissertation.

4.2 Inter-Cluster Synchronization

In this section we will describe the implemented synchronization between the 16 compute clusters that constitute the MPPA-256 processor.

Similarly to the last section, this one will start with a description and analysis of the communication method, and finish with a description of the developed synchronization algorithm.

4.2.1 The Communication Method

As mentioned in section 2.1.3.4, inter-cluster communication is done trough a set of special file descriptors that use the processor's NoCs in different ways.

For the propose of this dissertation we decided to use the `portal` connector. This connector directly implements the traditional remote memory access (RMA) operation called `PUT`, where a cluster can write in a specific memory area of another cluster. When using this connector, the reader cluster will reserve a memory area where any writer cluster can write, with an arbitrary offset that is chosen by each one of the writer clusters at the instant of writing[22].

The only way to interact with this type connector is with the adapted version of the POSIX AIO operations that are provided by Kalray. With the `mppa_aiocb_t` struct we can configure

the local memory area where remote processes will write, we also can install a callback function, using `mppa_aiocb_set_callback` and the `SIGEV_CALLBACK` flag, that will notify the reader process when a certain user defined message notification count is reached, this count can be set by using the `mppa_aiocb_set_trigger` function [22]. This means that if the notification count is, for example, set to 5 than the callback on the reader cluster will only be triggered when other clusters write to this connector at least 5 times.

To better understand the temporal behavior of this communication, we did a few tests to measure the message delays between the various clusters. We took advantage of the synchronized TSC to make these measurements.

Table 4.3: Results of the latency experiment with a portal connector

Cluster ID	Average Delay M⟶S (*ns*) with clusters of same parity	Average Delay S⟶M (*ns*) with clusters of same parity	Average Delay M⟶S (*ns*) with clusters of oposite parity	Average Delay S⟶M (*ns*) with clusters of oposite parity	Average Delay Asymmetry with clusters of same parity (*ns*)	Average Delay Asymmetry with clusters of oposite parity (*ns*)
0	3139,69	3587,00	1434,07	5442,44	448,94	4008,37
1	3156,88	3577,95	5091,59	1793,96	422,70	3297,62
2	3451,92	3336,94	1679,94	5239,91	190,90	3559,99
3	3442,26	3327,26	5284,81	1602,85	190,90	3681,95
4	3288,65	3512,05	1558,89	5383,64	183,36	3824,74
5	3278,32	3449,60	5158,56	1703,38	223,47	3455,18
6	3317,12	3416,16	1396,24	5477,48	203,19	4081,24
7	3319,81	3421,47	5178,88	1693,29	200,30	3485,59
8	3174,37	3570,06	1445,93	5425,31	399,85	3979,38
9	3176,57	3410,71	5252,81	1613,59	395,31	3639,20
10	3414,77	3319,63	1663,61	5208,92,	176,17	3545,50
11	3410,76	3319,24	5176,35	1608,91	174,32	3667,44
12	3269,69	3466,79	1526,22	5345,90	244,51	3819,69
13	3259,15	3478,10	5133,93	1738,66	253,78	3395,27
14	3109,40	3635,76	1602,95	5272,14	199,93	3669,19
15	3334,15	3398,10	5216,71	1666,56	192,74	3549,86

The tests were done with a master slave setup by exchanging 10000 pairs of messages between the master and all slaves, a total of 300000 messages for each experiment. This was done 16 times so that every cluster could assume the role of master. Table 4.3 shows the results of these experiments. An unexpected observation was made when analyzing these results, delay asymmetry between clusters with identifiers of different parity have a large systematic asymmetry of around 1500 cycles ($3,75\mu s$) that does not exist between clusters with IDs of the same parity. For example, a master/slave exchange between cluster 0 and cluster 1 will have this large delay asymmetry, that is always present, in addition to the smaller variable asymmetry. On the other hand, a master/slave exchange between cluster 0 and cluster 2 will not have this large systematic asymmetry and will only present the smaller variable asymmetry much closer to zero. The results for these specific examples can be seen in Figure 4.5, and are consistent with any other combination of clusters.

It was also verified that delays from *odd* numbered clusters to *even* numbered clusters were always larger than in the opposite direction, for example: $\delta_{1\to0} > \delta_{0\to1}$.

Initially it was considered that these asymmetries existed because messages were taking different routes trough the NoC. A solution to this would be to manually set the routes so that they were symmetric. To do this we can use the `mppa_get_unicast_route` function to calculate the routes during execution or by using a command line tool called `k1-nocencoderoute` that can be used to calculate the routes offline. During runtime we would need to call the `mppa_ioctl` function with the flag `MPPA_TX_SET_ROUTE` to establish the route before sending any message

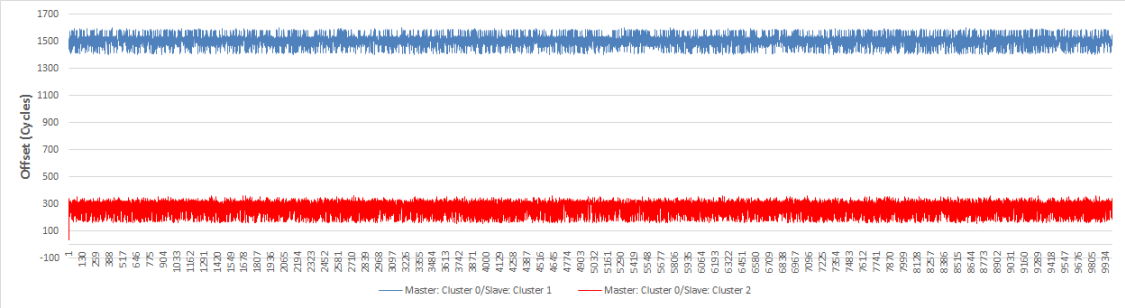


Figure 4.5: Example of the large systematic delay assymetry

[22]. This proved to be a fruitless endeavor that made little changes to the delay values. Therefore, we can conclude that these asymmetries are not caused by NoC path asymmetry.

It was also observed, by running the same test with the `Sync` connector, which uses the C-NoC as communication medium, that these large asymmetries are not constrained to the D-NoC and are also present on the C-NoC.

Even if we don't have any reasonable explanation for these large constant symmetries, they are backed by a large set of experimental data and will prove to be valuable information in the implementation of a master-slave synchronization algorithm.

4.2.2 The Synchronization Algorithm

The original idea for this level of the synchronization was to implement a fully distributed algorithm (Sections 2.2.4, 2.2.6, 2.2.7).When trying to implement some of these algorithms, some problems appeared that had significant negative impact in the synchronization's precision. The first problem arises from the fact that hardware broadcast is not available from the compute clusters and instead multicast is supported by software with a series of unicast messages. This introduces large timing differences between the clusters that receive the messages, differences that stem from unknown processing delays between the various unicast messages. This is a large problem for some of these algorithms since they use broadcast as its way to communicate and assume that all processes can observe the communication medium simultaneously. The second problem is the lack of precision of the reception notifications, this means that if the processes are closely synchronized all multicast messages received by a certain cluster will only activate one callback function making it impossible to measure the clock offset with and between the message senders by taking local timestamps of the arrival times. This would be a problem for a large number distributed algorithms but would not affect others that implement message staggering (Section 2.2.7).

These problems do not make it impossible to implement a fully distributed algorithm, but they will hinder their performance causing lower clock precision.

The purpose of choosing one of these algorithms was to increase the system's fault tolerance, but given the contained nature of our system we decided that it would not be necessary to consider overly strict fault models (Byzantine faults, Performance/Timing faults, etc.) and focused on

a simpler crash failure model. That being said, a centralized master-slave algorithm combined with a leader election would achieve the same degree of fault tolerance. On the other hand, it is obvious that this solution is less transparent to the synchronization because it would need dedicated election moments that would halt synchronization whenever a master crash happens. But since in our system failure probability is very small, the negatives of this approach are outweighed by the increase in precision.

Having reached these conclusions, we decided to also implement the IEEE 1588 standard PTP algorithm (Section 2.2.3) at cluster level. Unlike what happened with the intra-cluster synchronization, we will need to implement the full protocol since there is no shared memory environment for the synchronization.

This synchronization is entirely supported by the already mentioned portal connector, and a single "file" (`const char *portal = /mppa/portal/[0..15]:4;`) can be used to send and receive all messages by correctly multiplexing the target cluster with the `mppa_ioclt` function with the `MPPA_TX_SET_RX_RANK` request code.

With this synchronization approach we, again, face the problem of delay asymmetry. In the next section we will tackle this problem and propose a solution to minimize it.

4.2.3 Delay Asymmetry Correction

As shown in section 4.1.3, delay asymmetry in master-slave protocols can be a large source of error. Therefore, implementing delay asymmetry correction can greatly increase the synchronization's precision. The correction that we will be applying to the inter-cluster synchronization is divided in two stages.

This first stage only applies to clusters numbered with IDs of opposite parity to the master's ID, and it exists to compensate for the large systematic asymmetry between clusters with IDs of opposite parities that we talked about in Section 4.2.1. For this purpose, we implemented a constant delay compensation (Δ) with a value of the average measured delay asymmetry between these clusters. With this, delay asymmetries with opposite parity clusters decrease to much lower values that alternate much closer to zero just like the clusters that have the same parity as the master.

if parity (clusterID) = parity (masterID) **then**
    $\theta = (T_2 - T_1)/(T_4 - T_3)$;
else if parity (clusterID) = *odd* **then**
    $\theta = (T_2 - T_1)/(T_4 - T_3 + \Delta)$;
else
    $\theta = (T_2 - T_1 + \Delta)/(T_4 - T_3)$;
end if

Figure 4.6: Offset estimation with the constant delay compensation

In the second stage of the correction we apply a very similar filter to the one described in section 4.1.3. The one implemented in this level (Figure 4.7) differs from the one already used

only when a sample is not accepted by the filter condition. Whereas in the first implementation these samples would be discarded and nothing more would be done, in this new version, a counter is incremented every time this happens and every time a sample is accepted the counter is reset, but if this does not happen before it reaches a predefined ceiling value then the filter limits will be slightly increased in order to make it easier for new samples to be accepted. The counter's ceiling value is not a constant value, instead it is proportional to the number of accepted rounds during the algorithm's lifetime, the goal of this change is to increase the number of clock updates in the earlier stages of the algorithm's execution, avoiding long duration locks and decreasing the convergence times.

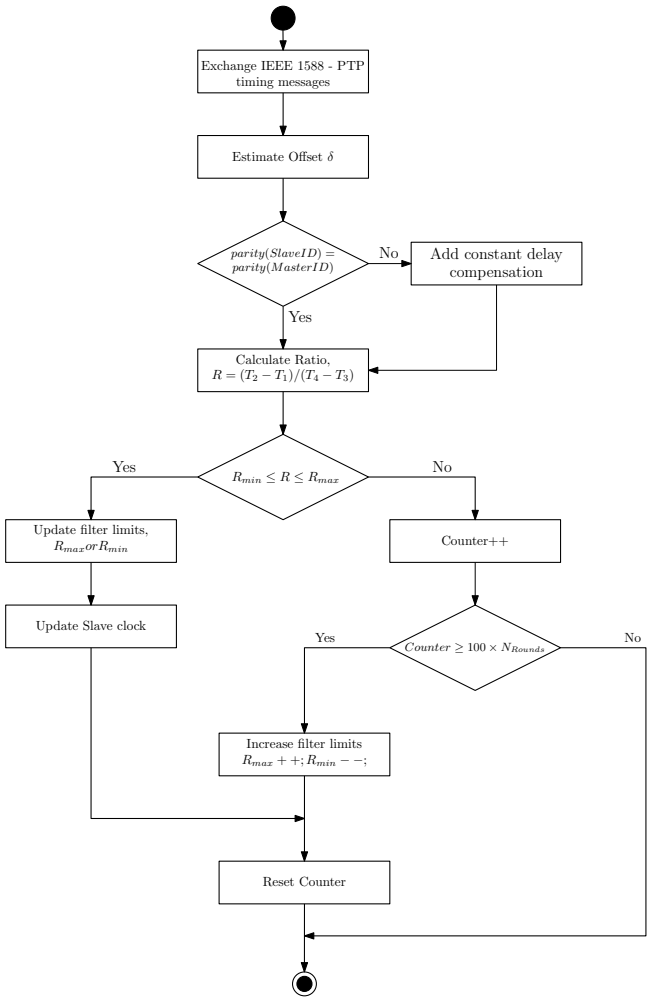


Figure 4.7: Flowchart of the adapted DAC model for the inter-cluster synchronization

This change arose from the first sets of experimental data on the inter-cluster synchronization. It was observed that in some experiments a cluster would only update its clock 1 or 2 times and than stop doing it for long periods of time, causing the larger initial offsets to extend during these periods. This new process that might increase the filters limits in the initial stages of the algorithm's execution eliminated this problem in any of the subsequent experiments.

4.3 Code Structure

The original plans for this dissertation was to develop some sort of kernel-level module that would implement the desired clock synchronization. Unfortunately, because of a lack of any relevant information and support to develop in the kernel of Kalray's proprietary NodeOS, we decided to implement our synchronization at user-level. But the MPPA's user-level cannot and should not be compared to user level programming on, for example, Linux. This because NodeOS is very simple and minimalist by comparison to the Linux kernel, and because nothing but the user code and the NodeOS kernel is actually running the cluster's PE cores, since management related tasks and the NodeOS hypervisor run on the RM core. This makes programming for the MPPA an experience much closer to developing firmware for an embedded platform.

With this in mind we decided to develop two algorithms, that only share a few variables and general use functions, for both inter and intra-cluster synchronization that can be included in other user applications independently. The source code was divided in six files, three headers and three source files. The full source code and doxygen documentation can be found in annex B and [39] respectively.

Whereas each of the synchronizations can be used stand-alone, if we choose to deploy both synchronizations at the same time a small problem will arise, because NodeOS only supports one POSIX timer per-core. Therefore, the intra-cluster synchronization has to be coordinated by the inter-cluster synchronization, and it will only be able to run with a period that equals any integer multiple of the inter-cluster synchronization's period. Although this configuration limits the possible periods for the intra-cluster synchronization it adds an advantage in guaranteeing no interference between the two synchronizations.

Notice that this limitation is only present at PE0, the timers of the other PE cores are free to be used by a user application.

In order for the system to work in this *merged* configuration the user will need to set the `INTERNAL_FLAG` to one, this flag is defined in the `common.h` header file.

Another important feature of our implementation if the fact that each of the PE cores, except for PE0, of any cluster can be running a user defined function. This is accomplished by using the desired function as the argument of the initialization routines of each synchronization. The only requirement is that at the start of this function the local clock is initialized with the following line of code: `init[__k1_get_cpu_id()] = __k1_read_dsu_timestamp();`.

Chapter 5

Evaluation of the Synchronization

In this chapter we analyze the exported results of the synchronizations to evaluate its quality.

5.1 Intra-Cluster Synchronization

5.1.1 Data export method

Since the synchronization that was implemented at this level greatly resembles the one in [11], we decided to take a similar approach to the one used there to evaluate its quality. Therefore, what we export from the processor are not clock values, instead we export the offset values calculated by the PTP algorithm and do it every time a correction is to be applied to any of the cluster's cores.

In reality, we export two additional values along with the clock offset. One of them is a counter that is incremented every time a PTP round is finished, this allows us to temporally characterize the results, since the synchronization rounds have a constant period we can observe how much time it takes for the algorithm to converge. The last value is the core id of the slave that was involved in this specific synchronization round, without this none of the exported data would have any meaning at all.

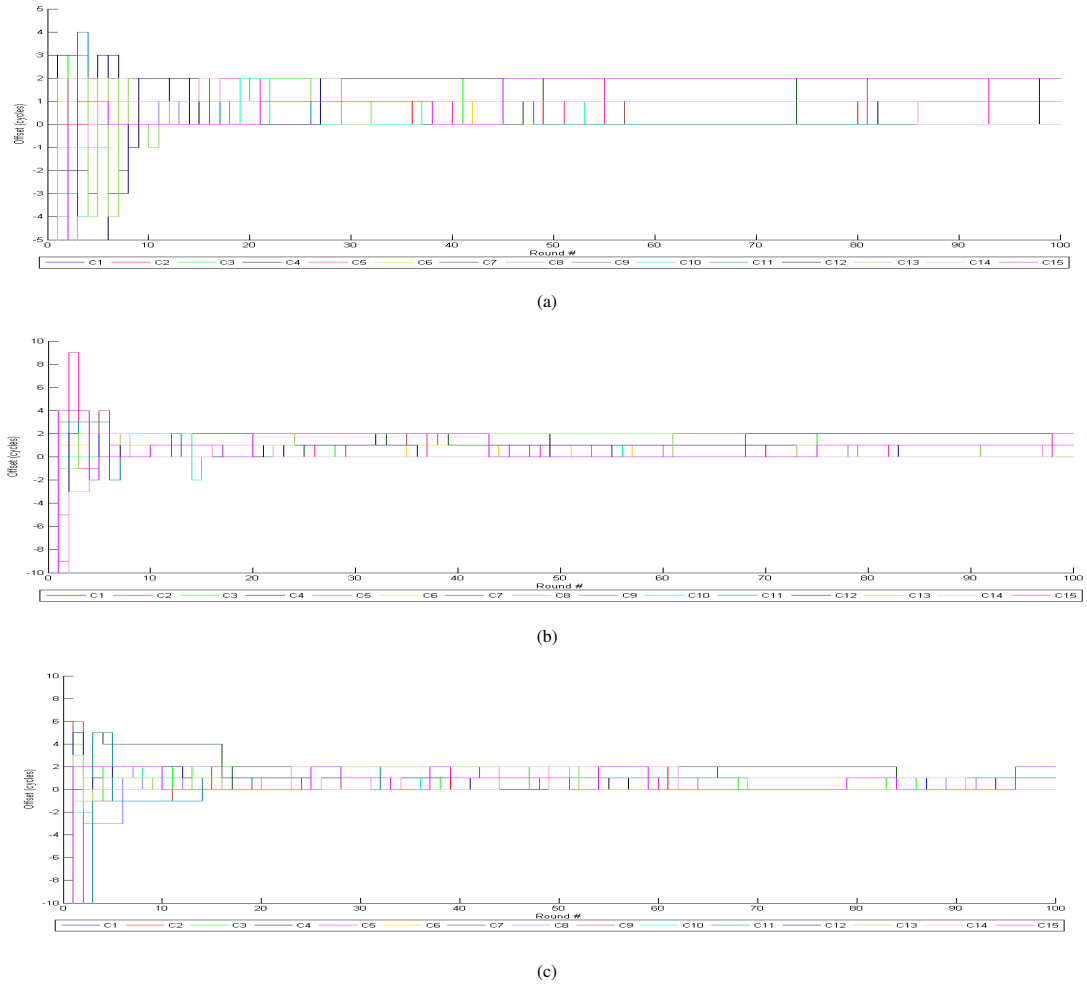
These values allow us to build a good view of synchronization's progression in each one of the cluster's cores.

5.1.2 Results

In order to get the all necessary data for a careful evaluation we ran three different experiments for the internal synchronization algorithm. In each experiment, the algorithm ran for a duration of 8 hours, and with different synchronization periods (1s, 100ms, 10ms). The results of these experiments can be found in Figures 5.1 and 5.2.

As it can be seen from Figure 5.1, the first few successful synchronization rounds are sufficient for the offset to converge around zero. This is not unsurprising since all clocks share the same hardware clock source, therefore there is no clock drift to correct. Instead, the effect of the synchronization is to correct the initial offset, which is too large in magnitude to be shown in the

graphs and is always negative because the master's clock is initialized before any of the slaves start their clocks. Another effect of the algorithm is to progressively reduce the clock read error thanks to the implemented DAC filter which will only accept better rounds to be accumulated with the offset correction value β_N, it is noticeable that the first iteration of the algorithm is nearly enough to synchronize the clocks and that the following rounds slowly reduce the remaining read error that is caused by the delay asymmetry.



(a)

(b)

(c)

Figure 5.1: First 100 Rounds of the intra-cluster synchronization results. (a) Experiment 1, $T = 1s$. (b) Experiment 2, $T = 100ms$. (c) Experiment 3, $T = 10ms$.

In Figure 5.2 we can see that after the initial period when the clocks are converging, they stay closely synchronized with offsets ranging from 0 to 2 clock cycles (0-5ns) during the remainder of the algorithm's lifetime. Another observable phenomenon in this figure is the fact that the frequency of accepted clock updates sharply decreases overtime. This is another effect of the implemented DAC filter which makes it increasingly difficult to find samples that respect the tighter quality factors. An important consideration regarding the three experiments comes from this decreasing frequency of updates, because it makes the implementation with the larger synchronization period a more efficient one, since the percentage of accepted rounds for each one of the

experiments is of $1,33\%$ for experiment 1, $0,18\%$ for experiment 2, and $0,037\%$ for experiment 3.

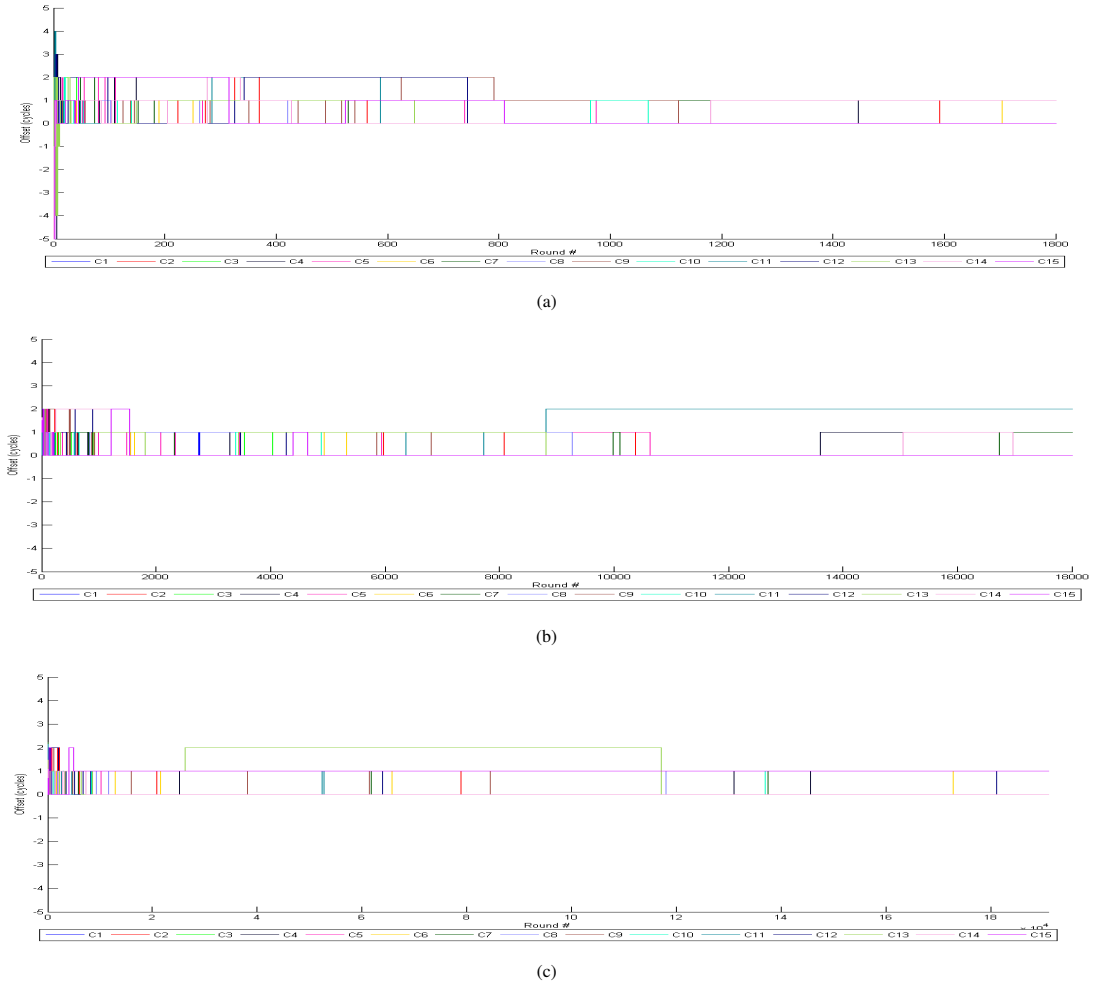


Figure 5.2: Intra-cluster synchronization results. (a) Experiment 1, $T = 1s$. (b) Experiment 2, $T = 100ms$. (c) Experiment 3, $T = 10ms$.

As expected, different synchronization periods don't have any effect on the synchronization's precision in steady state, this is an obvious result since there is no drift between the clocks. Furthermore, differences in the number of rounds that are necessary to achieve synchrony are only visible in the case of $T = 1s$, where we notice a slight increase in the number of rounds needed. Therefore the only relevant contrast between the three experiments is also an obvious one, the absolute time of convergence.

This means that the choice of synchronization period will be a compromise between overhead and the absolute convergence time, since smaller periods mean more messages exchanged but faster convergence.

5.2 Inter-Cluster Synchronization

5.2.1 Data export method

To evaluate the inter-cluster synchronization we devised a significantly different methodology to obtain the relevant data for our analysis from the one described on Section 5.1.1. The necessity for a different method arose from the fact that the PTP calculated offset can not be used to assess the quality of our synchronization method since we explicitly change the estimation made by the standard PTP algorithm with our constant compensation to correct delay asymmetry.

Our new approach takes advantage of the TSC's synchronization guarantee (Section 3.2.2.3) to determine the clock offset based on sampled clock values.

This sampling process is coordinated by the I/O subsystem. When a sampling period expires, the I/O will start a master/slave barrier with all 16 clusters. This barrier is implemented with the Sync connector and it exploits the available hardware broadcast to make all clusters sample their clocks at very close instants of time.

Unlike the `Portal` connector, the `Sync` connector uses the MPPA's C-NoC and messages can't carry data. The receiving (Rx) process of a `Sync` connector will setup a 64-bit word that can be OR'ed by the Transmitting (Tx) processes, with a value of their choosing. When this 64-bit word reaches -1, the Rx will unblock by successfully returning from its `mppa_read` call [22]. Therefore, this connector is ideal to implement barrier synchronization where a process, in this case the I/O subsystem, waits for all other processes (compute clusters) and then will broadcast a message to all of them so that they can sample their clocks at the arrival of this message.

In addition to the clock samples we export two extra values. One of them is the ID of the cluster where the clock sample was taken, the other one is the TSC value that was used to create the clock sample (Expression 3.1).

With these values and by taking advantage of the assumption that all TSCs are synchronized, we can correctly estimate the real offset between the master's clock and all slaves, trough expression 5.1 (see Figure 5.3).

$$\theta_{S \to M} = C_S - (C_M - (TSC_M - TSC_S)) = C_S - (C_M - \Delta_{TSC}) \tag{5.1}$$

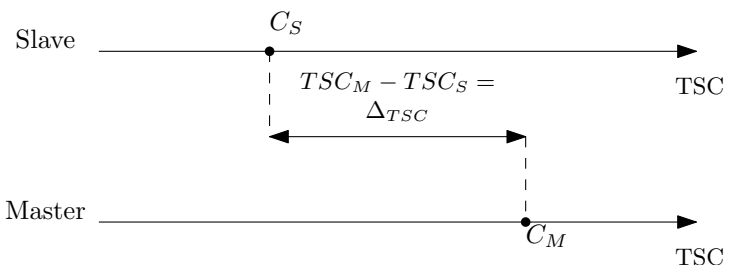
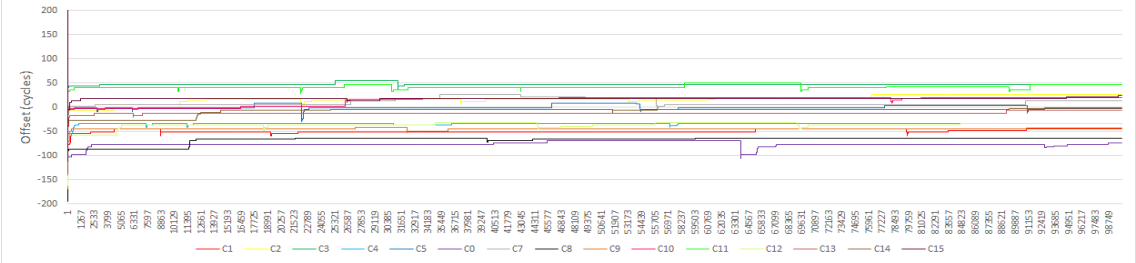


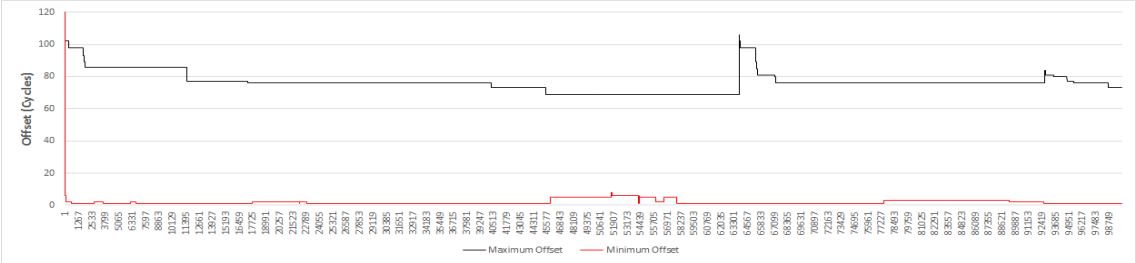
Figure 5.3: Offset calculation for the evaluation of the inter-cluster synchronization

5.2.2 Results

To assess the inter-cluster synchronization's quality, we carried out a set of three experiments. For each of them, the algorithm ran for a duration of 5 hours, with different master clusters (Cluster 6 for experiment 1, Cluster 14 for experiment 2, and cluster 5 for experiment 3), and with the same synchronization period of 10 ms. The results of these experiments can be seen in Figures 5.4, 5.5, and 5.6.



(a)



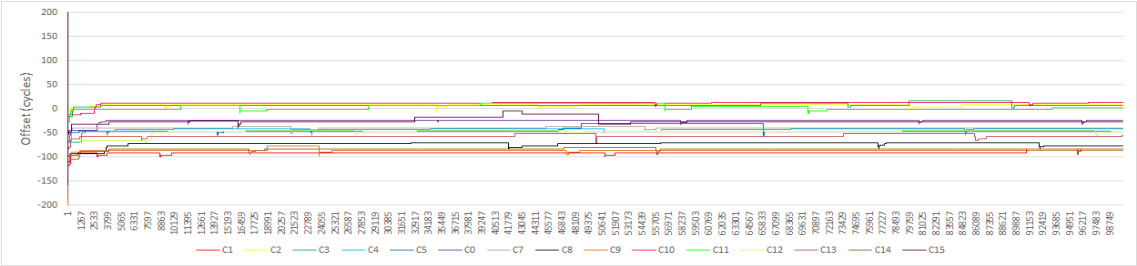
(b)

Figure 5.4: Experiment 1. Master: Cluster 6 (a) Offsets between each slave and the master cluster. (b) Absolute maximum and minimum offsets during the experiment

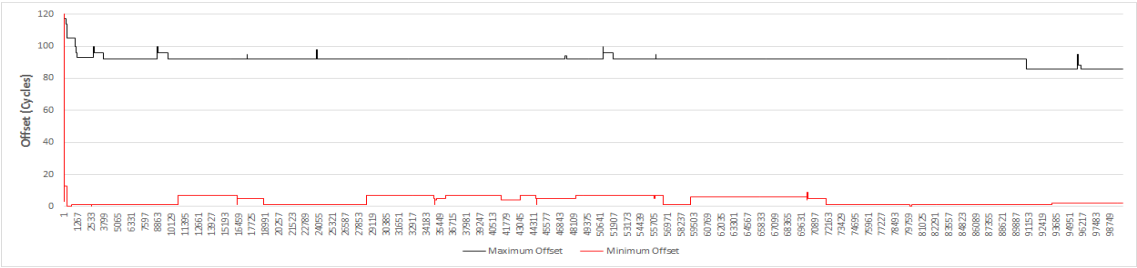
As it happened with the intra-cluster synchronization, the first iteration of the algorithm drastically reduces the offset between any of the slave clusters and the master, so much so that the initial offsets cannot be seen in the figures because of the large difference in magnitude.

Clock drift is not present since the whole system shares the same clock source. Therefore, the main effects of the algorithm are the offset correction and gradual reduction of the read error thanks to the implemented DAC filter, again, this was also true for the intra-cluster synchronization. But, in contrast, read errors at this level are considerably larger than inside the clusters, since communication delays and delay asymmetry are substantially larger, and each cluster has its own private TSC.

As a result of these larger errors, the steady state precision of the synchronization algorithm is noticeably lower than inside each cluster. Stable offsets can range from just a few clock cycles, 0-5 cycles ($0 - 12,5ns$), up to 100 cycles ($250ns$), which corresponds to about twice the amount of time it takes to read a value from the local TSC (Section 3.2.2.1). Maximum and minimum offsets for each experiment can also be found in Figures 5.4 (b), 5.5 (b), and 5.6 (b).

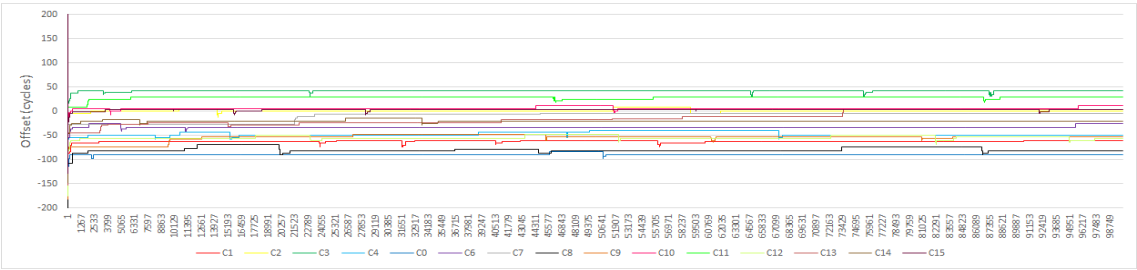


(a)

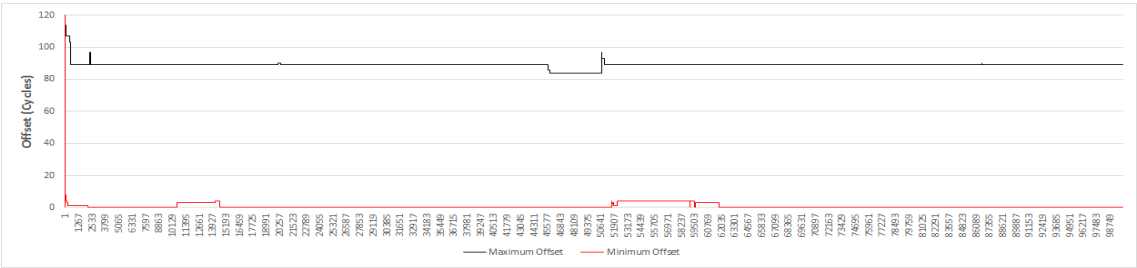


(b)

Figure 5.5: Experiment 2. Master: Cluster 14 (a) Offsets between each slave and the master cluster. (b) Absolute maximum and minimum offsets during the experiment



(a)



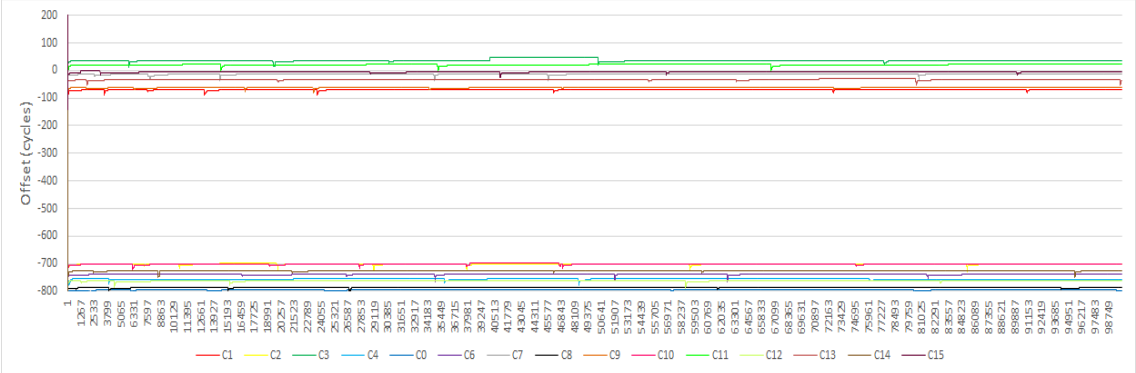
(b)

Figure 5.6: Experiment 3. Master: Cluster 5 (a) Offsets between each slave and the master cluster. (b) Absolute maximum and minimum offsets during the experiment

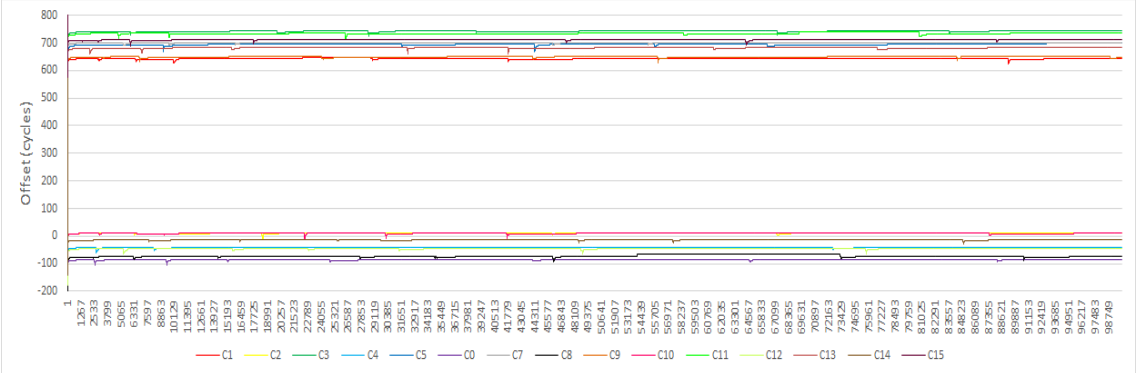
Additionally, as it can be seen from the various experiments, there are no significant precision differences between the various experiments with different masters, which confirms that the proposed implementation of an election algorithm is an appropriate solution to achieve fault tolerance.

Another similarity with the intra-cluster synchronization is in the gradual diminution of the frequency of clock updates because of the DAC filter making it increasingly harder to exceed the required quality factor. For experiment 1 only 0,17% of all synchronization rounds were applied, 0,18% in experiment 2, and 0,16% for the last experiment.

To verify the impact of the constant delay asymmetry correction, we decided to repeat some of the experiments already made but only with the DAC filter, and without applying the constant correction (Figure 5.7).



(a)



(b)

Figure 5.7: Experiments without the constant delay compensation (a) Master: Cluster 5. (b) Master: Cluster 6.

Just as expected, clusters with IDs of different parity from the ID of the master cluster present a much higher offsets, that range around 750 cycles (1875*ns*) which is about half of the average delay asymmetry, as it was anticipated from expression 4.2.

Chapter 6

Conclusions and Future Work

As it was said in Chapter 1, the main objective of this dissertation was to develop and implement a software based clock synchronization algorithm to achieve a consistent view of time throughout all cores in a many-core processor.

An extensive research of the related work was done with the objective of presenting a relevant review of the state of the art. We start this chapter by describing the evolution of multiprocessor systems that lead to the development of many-core processors.

Some of the most important and relevant architectural notions of many-core development were also presented, alongside with a deeper review of some specific many-core architectures.

Since clock synchronization was the main focus of this dissertation, it was also necessary to do a broad survey of different clock synchronization algorithms used in distributed systems.

The proposed synchronization uses two somewhat different implementations despite of the fact that both are based of PTP. Both levels of the synchronization can be deployed independently or can work together to achieve a common time base. Most of the differences arose because each one of the levels had to be implemented in two different programming models, the shared memory and message passing paradigms.

The choice and development of the algorithm was done after carrying out an important timing analysis of several key parameters of the target platform, Kalray's MPPA-256 architecture. Apart from being instrumental in allowing us to make an informed choice of what type of synchronization algorithm should be implemented, this analysis finds its relevancy in the fact that these types of platforms/architectures have been drawing more and more attention in real-time applications because of recent projects such as P-SOCRATES[40], CERTAINTY[41], and EMC^2[42].

Our work concluded with an evaluation of the quality of our implementation where we present an analysis of the algorithm's precision, convergence, and efficiency. The results of these experiments make us believe to have presented a good approach on the implementation of a clock synchronization algorithm in a many-core processor, and the problems that can arise during its implementation.

It's important to notice that the precision of any software approach will always be bounded by the precision of the underlying hardware. Furthermore, synchronization with dedicated hardware

resources will always be able to reach higher precisions. On the other hand, software based methods present much higher scalability, which is a key issue in many-core systems, since they do not cause an increase in area and power consumption of the processor.

6.1 Future Work

Because of the lack of relevant documentation our synchronization was implemented at the MPPA's user level, which might make it challenging for it to be included in other applications. The possibility of implementing the synchronization in NodeOS's kernel would be an improvement by increasing its transparency to the application programmer.

As for direct improvements to the proposed synchronization, the implementation of a rate correction method instead of an offset correction would create a completely monotonic clock by eliminating any possible discontinuities that might arise in the current implementation.

Furthermore, there is still the need of the implementation of an election algorithm to reach the desired level of fault tolerance. An interesting analysis would be to study the effect that change of master would have in the system's synchronization.

In alternative, another approach to fault tolerance would be to implement a synchronization algorithm capable of tolerating other types of faults, specifically byzantine faults, which would be a more suitable implementation for safety critical applications.

A more ambitious endeavor would be to implement our synchronization, or something similar to it, in a GALS type platform where multiple clock domains coexist, making rate correction a necessity instead of an improvement, and possibly forcing the synchronization to deal with clock drift if the various domains are generated from different oscillators, thus making some of the assumptions in this work inadequate.

Appendix A

Multiprocessor Operating Systems

> " Operating Systems represent the software foundation that enables applications to make use of the hardware resources of the computer in an efficient way. "[43]

Trough the years, several approaches to the implementation of multiprocessor operating systems have been designed. The most simple approach was to split the available memory in different partitions, one for each core, and give each core its own private operating system (OS). This model is rarely used anymore for several reasons, the main one being the fact that buffer caches need to be eliminated to avoid inconsistent results created by various cores accessing and caching the same block of data, causing a big performance penalty [44] .

Another OS model is known as *Asymmetric multiprocessing* (AMP). In this approach the various cores will have different assigned functionalities, in its simpler form, one of the cores will be the master and it will be running the OS and all other cores will be slaves doing user processes. The biggest problem with this model presents itself when the number of cores increases, because it creates a bottleneck at the master core. There are more complex types of AMP, usually to be deployed in heterogeneous hardware [43][44].

A third and the most common model for multiprocessor operating systems is called *Symmetric Multi-processing* (SMP). In this model, only one copy of the OS exists but it can be run by any of the processor cores. Every time a system call is made, the respective CPU where the call was made will try to run the necessary OS code. In its simpler form, OS code is protected by a lock, guaranteeing that the OS will only run by one of the cores at any given time. To avoid that this lock becomes a system bottleneck, the OS should be split into several critical sections with independent locks allowing the various cores to access different parts of the OS simultaneously [44].

A.1 SMP Linux

Linux implements a SMP system that features a monolithic kernel originally developed by Linus Thorvalds at the Helsinki University of Technology in 1992.

Linux only provides support to the symmetric multiprocessing approach and assumes cores with the same capabilities, but it has support for non-uniform memory systems (NUMA).

In SMP Linux each CPU has its own scheduler, the scheduling activity is complemented by a periodic task re-balancing between the various cores. At every scheduler tick, on each processor, the need for re-balancing is checked and initiated if need be. Cores can be divided into different scheduling domains that define in what scope the task re-balancing can be done. This process is done by a task stealing mechanism, in simple terms, when a CPU identifies the busiest core it will try to move some of its queuing task to himself. If not implemented with care, this re-balancing method can cause substantial contention when multiple cores try to move tasks from the busiest core.

Linux provides the user with an extensive and dynamic time management interface that is worth mentioning in the context of this dissertation. A detailed description of some of these features can be found at [11].

A.2 PikeOS

PikeOS is a paravirtualization RTOS developed by SYSGO AG to be used by multi-core processors in real-time and safety-critical applications. Its micro-kernel was designed with the concept of partitioning as main focus, which is important in safety-critical applications because of the need to eliminate inference between the various tasks.

PikeOS is ARINC 653 (Avionics Application Standard Software Interface) compatible since it provides complete time and space partitioning. Each partition is an independent application with a different memory space, and access to the I/O devices is also controlled by the OS. These two features grant spacial partitioning between applications.

A dedicated time slot is given to each system partition. Temporal partitioning is then ensured by an OS layer called the hypervisor. As mentioned before, this notion of time partition is inherited from ARINC 653, but pikeOS extends this concept by integrating multi-core support.

Each application can be mapped to a specific set cores. While a partition is running it has atomic access to all shared recourses, trough a method of time multiplexing. To guarantee this atomic access, all nodes need to have the same notion of time, which means a clock synchronization method is needed for its implementation in asynchronous systems.

PikeOS partitions can host a variety of different software such as LINUX, ARINC 653, POSIX, Android, RTEMS, AUTOSAR and Real-Time Java. The hypervisor model provided by the OS contains each guest inside its own virtual machine with a unique memory space and application set, which means that programs running on a VM are completely independent of programs at other VMs.

PikeOS offers two different configurations to support the SMP and AMP operating system models [13]. In Summary:

> " PikeOS together with system software forms a small minimal layer of trusted code and is therefore suited to safety-critical and secure certification standards "[13]

A.3 eMCOS

eMCOS is a RTOS developed by eSOL specifically targeting embedded many-core processors. It implements a distributed micro-kernel that can be used in processors with a wide disparity of core numbers because it does not rely on the existence of cache coherence protocols.

With eMCOS, an instance of the micro-kernel is deployed to every core in order to provide basic services such as inter-core communication and local thread scheduling. More advanced features are provided by server threads that are allocated to multiple cores. User applications can also allocated to threads that execute on multiple cores.

eMCOS schedules a thread based on its priority and core availability, but it also allows the user to directly say in which core the task should be executed.

This OS uses a patented scheduling algorithm based on two different types of schedulers that work simultaneously. The first guarantees real-time processing by scheduling the higher priority tasks to the respective cores, these tasks will then become the local threads with higher priorities and therefore are always executed when ready. For the other tasks a second scheduler is used, it tries to balance the load of all cores in order to achieve the lowest possible contention time for any thread (Figure A.1).

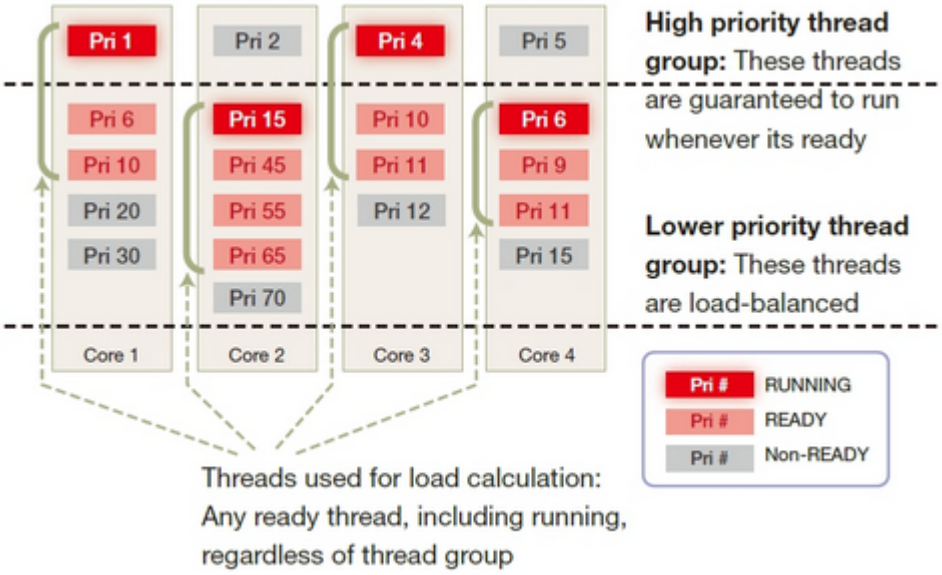


Figure A.1: eMCOS scheduling algorithm [12]

Threads in different cores can communicate via the micro-kernel's message-passing API. This API works as an abstraction layer between the developer and inner-workings of te communication mechanisms.

The eMCOS was designed to support any processor architecture, from single-core to many-core architectures, but given its relative infancy stage it only supports a small set of processors that include Kalray's MPPA-256 and Tilera's TILE-Gx8036.

Appendix B

Source Code

B.1 Common routines and variables

B.1.1 common.h

```
/** @file common.h
*  @brief Function prototypes for some general purpose routines common to both synchronizations
*
*  This contains the prototypes for some general purpose
*  routines common to both synchronizations
*  ,the parameters for each one of the cluster's clocks,
*  and some relevant constants and flags.
*
*  @author Filipe Monteiro (ee11120@fe.up.pt)
*  @bug No known bugs.
*/



#ifndef COMMON_H
#define COMMON_H

/**
* @brief Number of clusters of the MPPA
*/
#define CLUSTER_COUNT 16

/**
* @brief Number of cores inside each cluster
*/
#define CORE_COUNT 16



#ifndef MASTER
```

```
/**
* @brief Cluster id of the inter-cluster synchronization master
*/
#define MASTER 14
#endif

#ifndef INTERNAL_FLAG
/**
* @brief Flag to activate the intra-cluster synchronization.
*
* Flag to activate the intra-cluster synchronization, activated if = 1, deactivated if = 0,
* It's only necessary when both synchronizations are running.
*/
#define INTERNAL_FLAG 1
#endif

#ifndef INTERNAL_PERIOD
/**
* @brief Defines the intra-cluster synchronization period.
*
* Defines the intra-cluster synchronization period as
* a multiple of the period of the inter-cluster synchronization,
* It's only necessary when both synchronizations are running.
*/
#define INTERNAL_PERIOD 15
#endif

//Clock parameters
/**
* @brief Cluster ID
*/
volatile int rank, internal_period_counter;
/**
* @brief offset correction values for each of the clocks
*/
volatile long long beta[CORE_COUNT];
/**
* @brief rate correction values for each of the clocks
*/
volatile long long alpha[CORE_COUNT];
/**
* @brief initial values for each of the clocks
*/
volatile unsigned long long init[CORE_COUNT];

/**
* @brief number of synchronization rounds
*/
volatile unsigned long long round_k[CORE_COUNT];
```

```
/**
* @brief Read the current clock value and the underlying TSC value in a specific core
* @param core_id: Id of the core where the clock that is to be read is installed
* @param *tsc: Variable where the underlying TSC value that was used to create the
* clock sample is stored. For analysis purposes.
* @return The current clock value for the specified core
*/
inline unsigned long long get_clock(int core_id, unsigned long long *tsc);

/**
* @brief Disarms a POSIX timer
* @param timerid_disarm: TimerID
* @return If successful, function returns zero. Otherwise, it will return -1.
*/
int sync_timer_disarm(timer_t timerid_disarm);

/**
* @brief Arms a POSIX timer
* @param timerid_arm: TimerID
* @param period: Timer period in nanoseconds
* @return If successful, function returns zero. Otherwise, it will return -1.
*/
inline int sync_timer_arm(timer_t timerid_arm, unsigned long period);


#endif
```

B.1.2 common.c

```
/** @file common.c
*  @brief Common general purpose routines
*
*  Common general purpose routines that are
*  used throughout both implemented
*  synchronizations
*
*  @author Filipe Monteiro (ee11120@fe.up.pt)
*  @bug No know bugs.
*/

#include <time.h>
#include <stdio.h>
#include <HAL/hal/hal.h>
#include "common.h"

inline unsigned long long get_clock(int core_id, unsigned long long *tsc)
{
  __asm__ __volatile__ ("scall␣1059␣\n\t;;"
  : "+r" (*tsc)
```

```
  :
  : "memory", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r11", "r30", "r31", "r32", "r33", "r34",
  "r37", "r38", "r39",
  "r40", "r41", "r42", "r43", "r44", "r45", "r46", "r47", "r48", "r49", "r50", "r51", "r52", "r53", "r5
  "r56", "r57", "r58", "r59", "r60", "r61", "r62", "r63",
  "lc");
  return (alpha[core_id] * ((*tsc) - init[core_id]) ) - beta[core_id];
}


int sync_timer_disarm(timer_t timerid_disarm)
{
  struct itimerspec its;

  its.it_value.tv_sec = 0;
    its.it_value.tv_nsec = 0;
    its.it_interval.tv_sec = its.it_value.tv_sec;
    its.it_interval.tv_nsec = its.it_value.tv_nsec;


  //Disarm the timer
  if(timer_settime(timerid_disarm, 0, &its, NULL) <0)
  {
    return -1;
  }
  return 0;
}

inline int sync_timer_arm(timer_t timerid_arm, unsigned long period)
{
    struct itimerspec its;
    its.it_value.tv_sec = 0;
    its.it_value.tv_nsec = period;
    its.it_interval.tv_sec = its.it_value.tv_sec;
    its.it_interval.tv_nsec = its.it_value.tv_nsec;

    //Rearm the timer
    if(timer_settime(timerid_arm, 0, &its, NULL) <0)
    {
      return -1;
    }
    return 0;
}
```

B.2 Intra-Cluster Synchronization

B.2.1 internal_sync.h

```
/** @file internal_sync.h
*  @brief Function prototypes of the intra-cluster synchronization routines
*
*  This contains the prototypes for all the routines of the
* intra-cluster synchronization, some relevant configuration flags
*  and constants, and all variables shared between the cluster's cores.
*
*  @author Filipe Monteiro (ee11120@fe.up.pt)
*  @bug No known bugs.
*/


#ifndef INTERNAL_SYNC_H
#define INTERNAL_SYNC_H

#ifndef INTERNAL_SYNC_PERIOD
/**
* @brief Period for the intra-cluster synchronization timer
*/
#define INTERNAL_SYNC_PERIOD 1000000000
#endif

#ifndef INTERNAL_VERBOSE
/**
*
* @brief Flag to activate the verbose mode for the synchronization evaluation.
*
* Default value is zero, meaning the verbose mode is deactivated. Can be set to one
* in common.h by the user to activate the verbose. Verbose mode will allow the user
*   to verify the progression of the algorithm. Not recommended to be used with high
* synchronization frequencies because it will generate a very high amount of data.
*
*/
#define INTERNAL_VERBOSE 0
#endif

#ifndef INTERNAL_ANALYSIS
/**
*
* @brief Flag to activate the data export for the synchronization evaluation.
*
* Default value is zero, meaning the data export is deactivated. Can be set to one
* in common.h by the user to activate the data export.
*
*/
```

```
#define INTERNAL_ANALYSIS 0
#endif
/**
* @brief Thread id of the master thread
*/
pthread_t parent_thread;
/**
* @brief Thread id of all slave threads
*/
pthread_t threads[CORE_COUNT-1];
/**
* @brief Attrs for all slave threads
*/
pthread_attr_t attrs[CORE_COUNT-1];
/**
* @brief Timer ID of the intra-cluster synchronization POSIX timer
*/
timer_t internal_sync_timerid;

volatile unsigned long long round_count_master;


/**
* @brief Shared Memory to exchange timesatamps between master and slave
*/
volatile unsigned long long internal_tms2;
/**
* @brief Shared Memory to exchange timesatamps between master and slave
*/
volatile unsigned long long internal_tms3;
/**
* @brief Master Side Counter that stores the id of the next slave to be synchronized
*/
volatile unsigned int thread_counter;
/**
* @brief Flag for the termination of the synchronization
*/
volatile unsigned int kill_switch;
/**
*
* @brief Current high limit of the DAC filter for each of the cores.
* Initial value of 120%.
*
*/
volatile long internal_ratio_max[CORE_COUNT];
/**
*
* @brief Current low limit of the DAC filter for each of the cores.
* Initial value of 80%.
```

```
*
*/
volatile long internal_ratio_min[CORE_COUNT];

//POSIX Synchronization primitives
/**
* @brief Mutex for the synchronization between master and slave of the exchanged timestamps
*/
pthread_mutex_t  internal_lock;

/**
* @brief Mutex for the synchronization of the termination flag
*/
pthread_mutex_t kill_lock;

pthread_barrier_t test_barrier;

/**
*
* @brief Initializes the intra-cluster synchronization.
*
* This function is called by the master core (PE0) before
* starting the intra-cluster synchronization, it initializes
* all necessary variables, spawns the threads  in the other 15 cores,
* and starts the synchronization timer if necessary.
*
* @param task; Routine that will be running in the other 15 cores of the Cluster.
* @return If successful, function returns zero. Otherwise, it will return -1.
*
*/
int init_internal_sync(void *(*task)(void*));
/**
* @brief POSIX signal Handler
*
* Slave side synchronization routine triggered by the reception
* of a POSIX signal setup by init_internal_sync()
*
* @param sig: Signal id number
*/
void notify_signal_handler(int sig);
/**
* @brief POSIX timer callback function
*
* Callback function that is triggered by the expiration of the synchronization timer
*
* @param arg: Sigval union with relevant information about the timer
*/

void internal_sync_timer(union sigval arg);
```

```
/**
* @brief Slave thread routine
*
*  Small test routine for the slave core that initializes the local clock
*/
void* slave(void* args);
/**
*  @brief Finishes all slave threads and stops the intra-cluster synchronization
*/
void finish_internal_sync(void);


#endif
```

B.2.2 internal_sync.c

```
/** @file internal_sync.c
*  @brief Intra-cluster synchronization routines
*
*  This contains the implementation for all the routines of the
* intra-cluster synchronization.
*
*  @author Filipe Monteiro (ee11120@fe.up.pt)
*  @bug No known bugs.
*/

#include <pthread.h>
#include <time.h>
#include <signal.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <HAL/hal/hal.h>
#include <event_as.h>
#include "common.h"
#include "internal_sync.h"

int init_internal_sync(void *(*task)(void*)) //Master Side
{
  struct sigevent timer_evt;
  int core =2, i=0;
  //Setup the notify signal
  struct sigaction sigusr1;
  int status=0;
  sigusr1.sa_flags = SA_RESTART;
  sigusr1.sa_handler = notify_signal_handler;
  sigemptyset(&sigusr1.sa_mask);
  status = sigaction(SIGUSR1, &sigusr1, NULL);
  if (status == -1)
  {
```

```
  if (INTERNAL_VERBOSE !=0)
    printf("%d␣-␣Error␣installing␣signal\n", __k1_get_cpu_id());
  return -1;
}

parent_thread = pthread_self();
pthread_mutex_init(&internal_lock, NULL);
pthread_mutex_init(&kill_lock, NULL);
pthread_barrier_init(&test_barrier, NULL, CORE_COUNT);

//init couters and flag
round_count_master=0;
thread_counter = 0;
kill_switch = 0;

//Initialize the DAC filter limits for every core
for(i=0; i<CORE_COUNT; i++)
{
  internal_ratio_max[i] = 120;
  internal_ratio_min[i] = 80;
}

//Spawn the multiple slave threads
for(i=0; i<CORE_COUNT-1; i++)
{
  if (pthread_attr_init(&attrs[i]) != 0){
    if(INTERNAL_VERBOSE != 0)
      printf("error␣on␣attr␣init\n");
    return -1;
  }
  if (pthread_attr_setaffinity_np(&attrs[i], sizeof(unsigned int), &core) != 0){
    if(INTERNAL_VERBOSE!=0)
      printf("error␣setting␣core␣affinity\n");
    return -1;
  }
  if (pthread_create(&threads[i], &attrs[i], task, (void*)&parent_thread) !=0)
  {
    if(INTERNAL_VERBOSE!=0)
      printf("error␣creating␣thread\n");
    return -1;
  }
  core=core<< 1;
}

if(INTERNAL_FLAG == 0)
{
  for(i=0; i<CORE_COUNT; i++)
  {
    alpha[i]=1;
```

```
      beta[i]=0;
      init[i]=0;
      round_k[i]=0;
    }
    //Setup and start the POSIX timer for the synchronization
    //Sigevent struct for timer_create
    timer_evt.sigev_notify=SIGEV_CALLBACK;
    timer_evt.sigev_signo=0;
    timer_evt.sigev_value.sival_ptr=&internal_sync_timerid;
    timer_evt.sigev_notify_function = (void*) internal_sync_timer;

    if (timer_create(CLOCK_MONOTONIC, &timer_evt, &internal_sync_timerid) <0)
    {
      if(INTERNAL_VERBOSE!=0)
        printf("Error␣creating␣internal␣sync␣timer\n");
      return -1;
    }

    if (sync_timer_arm(internal_sync_timerid, INTERNAL_SYNC_PERIOD) < 0)
    {
      if(INTERNAL_VERBOSE!=0)
        printf("Error␣arming␣internal␣sync␣timer\n");
      return -1;
    }
  }
  if(INTERNAL_VERBOSE == 1)
    printf("Starting␣Internal␣Sync␣at␣cluster␣%d\n", rank);

  return 0;
}

void notify_signal_handler(int sig) //Slave Side
{
  unsigned long long tms1, aux_tms1, tms4, dummy, offset;
  nodeos_event_set event_in;
  int coreid = __k1_get_cpu_id();
  char str[100];
  signed long long t2_1, t4_3, ratio;


  //synchronization message exchange
  aux_tms1 = get_clock(coreid, &dummy);
  nodeos_event_send(parent_thread, NODEOS_EVENT_1); //Send Sync message
  tms1 = get_clock(coreid, &dummy);
  nodeos_event_receive(NODEOS_EVENT_0, NODEOS_EVENT_ANY, &event_in);
   //Recieve Delay request message
  tms4 = get_clock(coreid, &dummy);
  tms1= (tms1 + aux_tms1)/2;
```

```
  pthread_mutex_lock(&internal_lock); //Invalidates local cache, shared
   variables will have to be read again from memory

  t2_1 = (signed long long)(internal_tms2-tms1);
  t4_3 = (signed long long)(tms4-internal_tms3);


  offset =  (long long)( t4_3 - t2_1)  / 2 ;

  ratio = abs(lround((100 * ((double) t4_3/t2_1)));

  /*---------------DAC filter----------------*/
  if( ratio>=internal_ratio_min[coreid] && ratio<=internal_ratio_max[coreid])
  {
    beta[coreid] += offset; //Offset Correction
    if(INTERNAL_VERBOSE!=0)
      printf("Cluster␣%d␣---␣Core␣%d␣-␣offset␣=␣%lld␣-␣ratio␣=␣%lld\n"
      , rank, coreid, offset, ratio);

    if(ratio < 100 && ratio>internal_ratio_min[coreid]){
      internal_ratio_min[coreid]=ratio;
    }
    else if (ratio > 100 && ratio<internal_ratio_max[coreid]){
      internal_ratio_max[coreid]=ratio;
    }
    else if (ratio==100)
    {
      internal_ratio_max[coreid]=100;
      internal_ratio_min[coreid]=100;
    }

    if(INTERNAL_ANALYSIS!=0)
    {
      sprintf(str,"%d,%d,%d,%lld\n", rank, coreid, round_k[coreid], offset);
      printf("%s", str);
    }
  }

  round_k[coreid]++;

  pthread_mutex_unlock(&internal_lock); //Purges the changes done in cache to the shared memory

  return;
}

void* slave(void* args) //Slave Side
{
  unsigned long long tms, tsc;
```

```
  int core_id = __k1_get_cpu_id();
  struct timespec t;
  t.tv_sec=1;
  t.tv_nsec=0;
  init[core_id] = __k1_read_dsu_timestamp(); //Clock initialization
  if(INTERNAL_VERBOSE!=0)
    printf("Cluster %d -- Core %d started - init %llu\n", rank, core_id, init[core_id]);
  while(1)
  {
    nanosleep(&t, NULL);
    pthread_mutex_lock(&kill_lock);
    if(kill_switch != 0){
      pthread_mutex_unlock(&kill_lock);
      break;
    }
    pthread_mutex_unlock(&kill_lock);
  }
  pthread_barrier_wait(&test_barrier);
  tms=get_clock(core_id, &tsc);
  if(INTERNAL_VERBOSE!=0)
    printf("%d - clock %llu - %llu --- BETA = %lld\n", core_id, tms, tsc,
     beta[core_id]);
  return;
}

void internal_sync_timer(union sigval arg) //Master Side
{
  nodeos_event_set event_in;
  unsigned long long aux_timestamp, dummy;


  if (INTERNAL_FLAG !=1)
  {
    if (sync_timer_disarm(internal_sync_timerid) < 0)
    {
      if(INTERNAL_VERBOSE!=0)
        printf("Error disarming timer: internal sync\n");
      return;
    }
  }


  if(thread_counter > 14)
    thread_counter=0;
  //Send notification to one of the slave cores
  if (pthread_kill(threads[thread_counter], SIGUSR1) != 0 )
  {
    if(INTERNAL_VERBOSE!=0)
      printf("Error sending signal to %d\n", thread_counter);
```

```
    sync_timer_arm(internal_sync_timerid, INTERNAL_SYNC_PERIOD);
    return;
  }

  //--------------------Sync-round---------------------
  nodeos_event_receive(NODEOS_EVENT_1, NODEOS_EVENT_ANY, &event_in); //Recieved sync message
  internal_tms2 = get_clock(0, &dummy);

  pthread_mutex_lock(&internal_lock); //Invalidates local cache
  aux_timestamp = get_clock(0, &dummy);
  nodeos_event_send(threads[thread_counter], NODEOS_EVENT_0); //Send delay request message
  internal_tms3 = get_clock(0, &dummy);
  internal_tms3 = (internal_tms3 + aux_timestamp)/2;
  pthread_mutex_unlock(&internal_lock); //Flushes local cache
  //------------End sync round------------
  round_count_master++;
  thread_counter++;
  if (INTERNAL_FLAG !=1)
  {
    //Rearm the synchronization timer
    if (sync_timer_arm(internal_sync_timerid, INTERNAL_SYNC_PERIOD) < 0)
    {
      if(INTERNAL_VERBOSE!=0)
        printf("Error␣rearming␣internal␣sync␣timer\n");
    }
  }
  return;
}


void finish_internal_sync(void)
{
  int i=0;
  unsigned long long tms, tsc;

  if(INTERNAL_FLAG==0){
    sync_timer_disarm(internal_sync_timerid);
    timer_delete(internal_sync_timerid);
  }

  //Triggers the kill flag, stoping the synchronization
  pthread_mutex_lock(&kill_lock);
  kill_switch = 1;
  pthread_mutex_unlock(&kill_lock);
  pthread_barrier_wait(&test_barrier);
  tms=get_clock(0, &tsc);

  if(INTERNAL_VERBOSE!=0)
    printf("0␣-␣clock␣%llu␣-␣%llu␣␣-␣␣BETA␣=␣%lld\n", tms, tsc, beta[0]);
```

```
  for(i=0; i<CORE_COUNT-1; i++)
  {
      pthread_join(threads[i], NULL);
  }

  pthread_mutex_destroy(&internal_lock);
  pthread_mutex_destroy(&kill_lock);
  pthread_barrier_destroy(&test_barrier);

  return;
}
```

B.3 Inter-Cluster Synchronization

B.3.1 external_sync.h

```
/** @file  external_sync.h
*  @brief Function prototypes of the inter-cluster synchronization routines
*
*  This contains the prototypes for all the routines of the
* inter-cluster synchronization, some relevant configuration flags
*  and constants, and all global variables necessary because of
* context changes during the synchronization (callback functions)
*
*  @author Filipe Monteiro (ee11120@fe.up.pt)
*  @bug No known bugs.
*/

#ifndef EXTERNAL_SYNC_H
#define EXTERNAL_SYNC_H

/**
* @brief Period for the inter-cluster synchronization timer
*/
#define EXTERNAL_SYNC_PERIOD 10000000 //10ms
/**
* @brief Constant compensation for the delay asymmetry
*/
#define DELTA 1550 //1466
/**
* @brief State of the PTP slave state machine
*/
#define SYNC 0
/**
* @brief State of the PTP slave state machine
*/
#define FOLLOW_UP 1
/**
```

```
* @brief State of the PTP slave state machine
*/
#define DELAY_RESP 2

#ifndef EXTERNAL_VERBOSE
/**
*
* @brief Flag to activate the verbose mode for the synchronization evaluation.
*
* Default value is zero, meaning the verbose mode is deactivated. Can be set to one
* in common.h by the user to activate the verbose. Verbose mode will allow the user
*   to verify the progression of the algorithm. Not recommended to be used with high
* synchronization frequencies because it will generate a very high amount of data.
*
*/
#define EXTERNAL_VERBOSE 0
#endif



/**
* @brief Timer ID of the inter-cluster synchronization POSIX timer
*/
timer_t external_sync_timerid;

/**
* @brief PTP timestamp
*/
volatile unsigned long long tms1;
/**
* @brief PTP timestamp
*/
volatile unsigned long long tms2;
/**
* @brief PTP timestamp
*/
volatile unsigned long long tms3;
/**
* @brief PTP timestamp
*/
volatile unsigned long long tms4;
/**
* @brief Recieve buffer for the portal connector
*/
volatile unsigned long long results;
volatile unsigned long long counter;
/**
* @brief File descriptor for the recieve portal connector
*/
```

```
volatile int cluster_tx_portal_fd;
/**
* @brief File descriptor for the transmit portal connector
*/
volatile int cluster_rx_portal_fd;

volatile int slave_counter, slave_state, external_round_count_master;

/**
* @brief Low limit of the DAC filter
*/
volatile unsigned long ratio_min;
/**
* @brief High limit of the DAC filter
*/
volatile unsigned long ratio_max;


typedef void* slave_task(void* args);

slave_task *s_task;

/**
* @brief Initializes the inter-cluster synchronization.
*
* This function is called by all clusters before
* starting the inter-cluster synchronization, it initializes
* all necessary variables,
* and starts the synchronization timer if necessary.
* It also initializes the intra-cluster synchronization
* in the case of the merged configuration.
*
* @param task; Routine that will be running in the other 15 cores of the Cluster.
* To be used in init_internal_sync() .
* @return If successful, function returns zero. Otherwise, it will return -1.
*/
int init_external_sync(void *(*routine)(void*));
/**
* @brief Recieve callback function for the Portal Connector - Master Side
*
* Callback function that is triggered by the arrival of
* a message trough the synchronization's
* portal connector.
*
* @param arg: Sigval union with relevant information about the connector
*/
void master_rx_callback(mppa_sigval_t arg);

/**
```

```
* @brief Recieve callback function for the Portal Connector - Slave Side
*
* Callback function that is triggered by the arrival of a message trough the synchronization's
* portal connector.
*
* @param arg: Sigval union with relevant information about the connector
*/

void slave_rx_callback(mppa_sigval_t arg);
/**
* @brief POSIX timer callback function
*
* Callback function that is triggered by the expiration of the synchronization timer
*
* @param arg: Sigval union with relevant information about the timer
*/

void external_sync_timer(union sigval arg);
/**
*  @brief Closes all communication connectors, stops the synchronization
* timer in the master, and calls finish_internal_sync()
* in case of merged configuration (If INTERNAL_FLAG == 1).
*/
void finish_external_sync(void);


#endif
```

B.3.2 external_sync.c

```
/** @file external_sync.c
*  @brief Inter-cluster synchronization routines
*
*  This contains the implementation for all the routines of the
* inter-cluster synchronization.
*
*  @author Filipe Monteiro (ee11120@fe.up.pt)
*  @bug No known bugs.
*/


#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <HAL/hal/hal.h>
#include <mppaipc.h>


#include "common.h"
```

```
#include "external_sync.h"
#include "internal_sync.h"

int parity(int id)
{
  return id%2;
}

int init_external_sync(void *(*routine)(void*))
{
  const char *portal = "/mppa/portal/[0..15]:4";
  static mppa_aiocb_t *aiocb_s;
  struct sigevent timer_evt;
  s_task=routine;
  tms1=0;
  tms2=0;
  tms3=0;
  tms4=0;
  results=0;
  ratio_min=80;
  ratio_max=120;
  slave_state=0;
  slave_counter=0;
  counter=0;
  external_round_count_master=0;
  if (MASTER == 0)
    slave_counter++;
  for(i=0; i<CORE_COUNT; i++)
  {
    alpha[i]=1;
    beta[i]=0;
    init[i]=0;
    round_k[i]=0;
  }
  init[0]=__k1_read_dsu_timestamp();
  if (EXTERNAL_VERBOSE != 0)
    printf("%d␣-␣init␣=␣%llu\n",rank, init[0]);
  cluster_tx_portal_fd = mppa_open(portal, O_WRONLY);
  if(cluster_tx_portal_fd == -1)
  {
    if (EXTERNAL_VERBOSE != 0)
      printf("Error␣opening␣IO␣portal␣connector\r\n");
    return -1;
  }

  cluster_rx_portal_fd = mppa_open(portal, O_RDONLY);
  if(cluster_rx_portal_fd == -1)
  {
    if (EXTERNAL_VERBOSE != 0)
```

```
      printf("Error␣opening␣IO␣portal␣connector\r\n");
    return -1;
  }

  mppa_aiocb_t results_portal_aiocb[1]={MPPA_AIOCB_INITIALIZER
  (cluster_rx_portal_fd, &results, sizeof(results))} ;

  if(rank==MASTER)
  {
    if(INTERNAL_FLAG==1)
    {
      internal_period_counter=0;
      init_internal_sync(s_task);
    }
    aiocb_s=results_portal_aiocb;
    mppa_aiocb_set_callback(results_portal_aiocb, master_rx_callback);
    mppa_aiocb_set_trigger(aiocb_s, 1);


    timer_evt.sigev_notify=SIGEV_CALLBACK;
    timer_evt.sigev_signo=0;
    timer_evt.sigev_value.sival_ptr=&external_sync_timerid;
    timer_evt.sigev_notify_function = (void*) external_sync_timer;

    if (timer_create(CLOCK_MONOTONIC, &timer_evt, &external_sync_timerid) <0)
    {
      if (EXTERNAL_VERBOSE != 0)
        printf("Error␣creating␣external␣sync␣timer\n");
      return -1;
    }

    if (sync_timer_arm(external_sync_timerid, EXTERNAL_SYNC_PERIOD) < 0)
    {
      if (EXTERNAL_VERBOSE != 0)
        printf("Error␣arming␣external␣sync␣timer\n");
      return -1;
    }
  }
  else
  {
    aiocb_s=results_portal_aiocb;
    mppa_aiocb_set_callback(results_portal_aiocb, slave_rx_callback);
    mppa_aiocb_set_trigger(aiocb_s, 1);
  }
  mppa_aio_read(aiocb_s);


  return 0;
}
```

```
void master_rx_callback(mppa_sigval_t arg)
{
  unsigned long long dummy, t;
  tms4=get_clock(0, &dummy);
  t=tms4;

  mppa_ioctl(cluster_tx_portal_fd, MPPA_TX_SET_RX_RANK, slave_counter);
  mppa_pwrite(cluster_tx_portal_fd, &t, sizeof(unsigned long long), 0);//Send Delay Responce msg
  slave_counter++;
  if(slave_counter==rank)
    slave_counter++;
  if(slave_counter>15)
  {
    if (rank!=0)
      slave_counter=0;
    else
      slave_counter=1;
  }
  external_round_count_master++;
  return;
}

void slave_rx_callback(mppa_sigval_t arg)
{
  unsigned long long dummy;
  unsigned long long t=get_clock(0, &dummy);
  signed long long t2_1, t4_3, ratio;
  mppa_aiocb_t *aiocb_s;
  aiocb_s=arg.sival_ptr;
  unsigned long long *res = (unsigned long long*)aiocb_s->aio_buf;
  long long offset=0;
  union sigval dummy_sigval;

  if (slave_state == SYNC)
  {
    tms2=t;
    slave_state=FOLLOW_UP;
    if (EXTERNAL_VERBOSE != 0)
      printf("%d␣recieved␣sync␣msg\n", rank);
  }
  else if (slave_state==FOLLOW_UP)
    {
      if (EXTERNAL_VERBOSE != 0)
        printf("%d␣recieved␣follow␣up␣msg\n", rank);
      tms1=results;
      mppa_ioctl(cluster_tx_portal_fd, MPPA_TX_SET_RX_RANK, MASTER);
      mppa_pwrite(cluster_tx_portal_fd, &t, sizeof(unsigned long long), 0); //Send Delay Request
```

```
  tms3=get_clock(0, &dummy);
  slave_state=DELAY_RESP;
}
else if (slave_state == DELAY_RESP)
  {
    if (EXTERNAL_VERBOSE != 0)
      printf("%d␣recieved␣delay␣responce␣msg\n", rank);
    tms4=results;
    if(parity(MASTER) == parity(rank))
    {
      t2_1 = (signed long long)(tms2-tms1);
      t4_3 = (signed long long)(tms4-tms3);
    }
    else if (parity(rank) == 1)
      {
        t2_1 = (signed long long)(tms2-tms1+DELTA);
        t4_3 = (signed long long)(tms4-tms3);
      }
      else if (parity(rank)==0)
      {
        t2_1 = (signed long long)(tms2-tms1);
        t4_3 = (signed long long)(tms4-tms3+DELTA);
      }


    offset = (long long) (t2_1 - t4_3)/2;

    ratio = abs(lround(100 * ((double) t4_3/t2_1)));

    /*--------------------DAC Filter--------------------*/
    if( (ratio>=ratio_min && ratio<=ratio_max))
    {
      if (EXTERNAL_VERBOSE != 0)
        printf("Cluster:␣%d␣-----␣offset␣=␣%lld␣------␣ratio␣=␣%lld\n", rank, offset, ratio);
      beta[0]=beta[0]+offset;
      counter=0;
      if(round_k[0] > 0)
      {
        if(ratio < 100 && ratio>ratio_min){
          ratio_min=ratio;
        }
        else if (ratio > 100 && ratio<ratio_max){
          ratio_max=ratio;
        }
        else if (ratio==100)
        {
          ratio_max=100;
          ratio_min=100;
        }
```

```
          }
          if(round_k[0]==0 && INTERNAL_FLAG==1)
            init_internal_sync(s_task);
          round_k[0]++;
        }
        else
          counter++;

        if(counter>100*round_k[0])
        {
          ratio_max=ratio_max+1;
          ratio_min=ratio_min-1;
          counter=0;
        }

/*--------------------Intra-cluster Synchronization--------------------*/

        if(INTERNAL_FLAG == 1 && round_k[0] > 0)
        {
          internal_period_counter++;
          if(internal_period_counter >= (INTERNAL_PERIOD/15) )
          {
            if (EXTERNAL_VERBOSE != 0)
              printf("start␣of␣internal␣sync␣%d\n", internal_period_counter);

            internal_sync_timer(dummy_sigval);
            internal_period_counter=0;
          }
        }
        slave_state=SYNC;
      }

  return;
}


void external_sync_timer(union sigval arg) //Master Side
{
  unsigned long long t=0;
  union sigval arg2;
  if (sync_timer_disarm(external_sync_timerid) < 0)
  {
    if (EXTERNAL_VERBOSE != 0)
      printf("Error␣disarming␣timer:␣external␣sync\n");
    return;
  }
  mppa_ioctl(cluster_tx_portal_fd, MPPA_TX_SET_RX_RANK, slave_counter);
  mppa_pwrite(cluster_tx_portal_fd, &t, sizeof(unsigned long long), 0); //Sync Message
  tms1=get_clock(0, &t);
```

```
  t=tms1;
  mppa_pwrite(cluster_tx_portal_fd, &t, sizeof(unsigned long long), 0); //Follow-up message


  if(INTERNAL_FLAG == 1){
    internal_period_counter++;
    if(internal_period_counter >=INTERNAL_PERIOD){
      if (EXTERNAL_VERBOSE != 0)
        printf("start␣of␣internal␣sync␣%d\n", internal_period_counter);

      internal_sync_timer(arg2);
      internal_period_counter=0;
    }
  }
  if (sync_timer_arm(external_sync_timerid, EXTERNAL_SYNC_PERIOD) < 0)
  {
    if (EXTERNAL_VERBOSE != 0)
      printf("Error␣rearming␣external␣sync␣timer\n");
  }
  return;
}

void finish_external_sync(void)
{
  if(rank==MASTER)
  {
    sync_timer_disarm(external_sync_timerid);
    timer_delete(external_sync_timerid);
  }
  mppa_close(cluster_rx_portal_fd);
  mppa_close(cluster_tx_portal_fd);
  if (INTERNAL_FLAG==1)
    finish_internal_sync();
}
```

References

[1] Harold S. Stone. *High Performance Computer Architecture*. Addison-Wesley Publishing Company, 1993.

[2] Borislav Nikolic. *Many-Core Platforms in the Real-Time Embedded Computing Domain*. Thesis, FEUP, 2015. URL: http://hdl.handle.net/10216/78996.

[3] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w TeraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, Jan 2008. doi:10.1109/JSSC.2007.910957.

[4] Max Baron. The single-chip cloud computer. *The Linley Group, Microprocessor Report*, 2010. URL: http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-article.pdf.

[5] D. Wentzlaff, P. Griffin, H. Hoffmann, Bao Liewei, B. Edwards, C. Ramey, M. Mattina, Miao Chyi-Chang, J. F. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, 2007. doi:10.1109/MM.2007.4378780.

[6] B. D. de Dinechin, R. Ayrignac, P. E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, 2013. doi:10.1109/HPEC.2013.6670342.

[7] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems principles and paradigms*, volume 2nd ed. Pearson Prentice Hall, Uper Saddle River, NJ, 2007. eng.

[8] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, 1990. doi:10.1109/2.58235.

[9] P. Sommer and R. Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *Information Processing in Sensor Networks, 2009. IPSN 2009. International Conference on*, pages 37–48, 2009.

[10] Geoffrey Werner-Allen, Geetika Tewari, Ankit Patel, Matt Welsh, and Radhika Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects, 2005. doi:10.1145/1098918.1098934.

[11] André dos Santos Oliveira. *Clock Synchronization for Moderm Multiprocessors*. Thesis, FEUP, 2015. URL: http://hdl.handle.net/10216/79624.

[12] eSOL. eMCOS, 2015. URL: http://www.esol.com/embedded/emcos.html#microkernel.

[13] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems, 2015.

[14] M. Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, 2007. doi:10.1109/N-SSC.2007.4785534.

[15] R. H. Dennard, F. H. Gaensslen, Hwa-Nien Yu, V. L. Rideout, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid State Circuits*, 9(5):256–268, 1974. doi:10.1109/JPROC.1999.752522.

[16] Andrs Vajda. *Chapter 2: Multi-core and Many-core Processor Architectures*, pages 9–43. Springer Publishing Company, Incorporated, 2011.

[17] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 278–287, 1992. doi:10.1109/ISCA.1992.753324.

[18] V. Carl Hamacher, Zvonko G. Vranesic, and Safwat G. Zaky. *Computer Organization*. McGraw-Hill International Editions, 1996.

[19] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1629575.1629579, doi:10.1145/1629575.1629579.

[20] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002. doi:10.1109/mm.2002.997877.

[21] B. D. de Dinechin, D. van Amstel, M. Poulhies, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation and Test in Europe Conference and Exhibition, 2014*, pages 1–6, 2014. doi:10.7873/DATE.2014.110.

[22] Kalray S.A. *MPPA® ACCESSCORE POSIX Programming Reference Manual*. Kalray S.A., 2015.

[23] Benoît Dupont de Dinechin, Yves Durand, Duco van Amstel, and Alexandre Ghiti. Guaranteed services of the noc of a manycore processor, 2014. doi:10.1145/2685342.2685344.

[24] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the Kalray MPPA®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013. URL: http://www.sciencedirect.

com/science/article/pii/S1877050913004766, doi:http://dx.doi.org/10.1016/j.procs.2013.05.333.

[25] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition*. CRC Press, Inc., 2010.

[26] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–58, 1989. URL: http://dx.doi.org/10.1007/BF01784024, doi:10.1007/BF01784024.

[27] NIST. Introduction to IEEE 1588, 2014. URL: http://www.nist.gov/el/isd/ieee/intro1588.cfm.

[28] John Edison (agilent). IEEE-1588 standard for a precision clock synchronization protocol for networked measurement and control systems -a tutorial-, 2005. URL: http://www.nist.gov/el/isd/ieee/upload/tutorial-basic.pdf.

[29] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, 1985. doi:10.1145/2455.2457.

[30] Jennifer Lundelius Welch and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988. URL: http://www.sciencedirect.com/science/article/pii/0890540188900430, doi:http://dx.doi.org/10.1016/0890-5401(88)90043-0.

[31] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *J. ACM*, 34(3):626–645, 1987. doi:10.1145/28869.28876.

[32] Rui Fan and Nancy Lynch. Gradient clock synchronization. *Distrib. Comput.*, 18(4):255–266, 2006. doi:10.1007/s00446-005-0135-6.

[33] Rodolfo M. Pussente and Valmir C. Barbosa. An algorithm for clock synchronization with the gradient property in sensor networks. *J. Parallel Distrib. Comput.*, 69(3):261–265, 2009. doi:10.1016/j.jpdc.2008.11.001.

[34] T. Herman and Zhang Chen. Best paper: stabilizing clock synchronization for wireless sensor networks. In *Stabilization, Safety, and Security of Distributed Systems. 8th International Symposium, SSS 2006. Proceedings, 17-19 Nov. 2006*, Stabilization, Safety, and Security of Distributed Systems 8th International Symposium, SSS 2006. Proceedings (Lecture Notes in Computer Science Vol. 4280), pages 335–49. Springer-Verlag, 2006.

[35] Renato E. Mirollo and Steven H. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM J. Appl. Math.*, 50(6):1645–1662, 1990. doi:10.1137/0150098.

[36] R. Leidenfrost and W. Elmenreich. Firefly clock synchronization in an 802.15.4 wireless network. *EURASIP Journal on Embedded Systems*, page 186406 (17 pp.), 2009. URL: http://dx.doi.org/10.1155/2009/186406, doi:10.1155/2009/186406.

[37] M. A. Rahman, T. Kunz, and H. Schwartz. Delay asymmetry correction model for master-slave synchronization protocols. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 1–8, 2014. doi:10.1109/AINA.2014.8.

[38] R. Exel. Mitigation of asymmetric link delays in IEEE 1588 clock synchronization systems. *IEEE Communications Letters*, 18(3):507–510, 2014. doi:10.1109/LCOMM.2014.012214.132540.

[39] Filipe Monteiro. Clock synchronization for many-core processors - companion website - doxygen, 2016. URL: https://paginas.fe.up.pt/~ee11120/dissertation/doxygen.

[40] Luís Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, and Michal Mardiak. P-SOCRATES: A parallel software framework for time-critical many-core systems. *Microprocessors and Microsystems*, 39(8):1190 – 1203, 2015. URL: http://www.sciencedirect.com/science/article/pii/S0141933115000836, doi:http://dx.doi.org/10.1016/j.micpro.2015.06.004.

[41] CERTAINTY: Certification of real time applications designed for mixed criticality. URL: http://www.certainty-project.eu/.

[42] *EMC*^{2}: Embedded multi-core systems for mixed criticality applications in dynamic and changeable real-time environments. URL: http://www.artemis-emc2.eu/.

[43] Andrs Vajda. *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 2011.

[44] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, 2007.